HZ BOOKS
华章教育
PEARSON

计 算 机 科 学 丛 书

《人月神话》作者最新力作
计算机科学大师探究设计原本

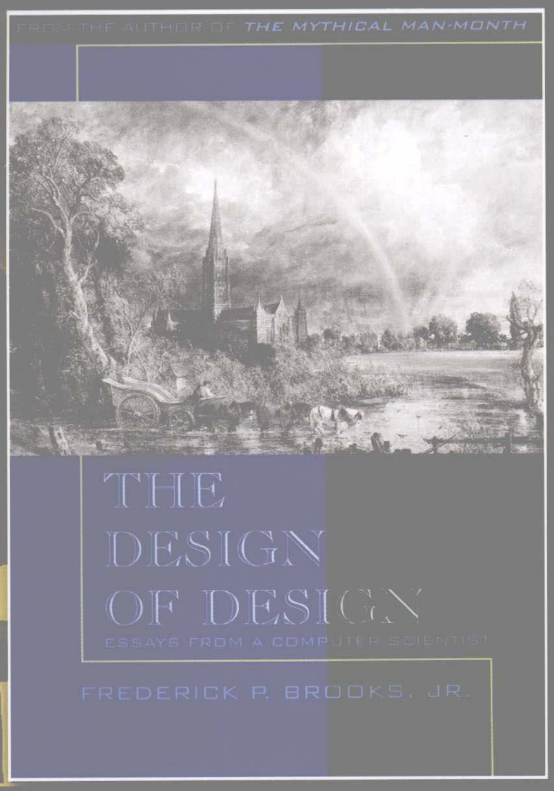
设计原本

计算机科学巨匠Frederick P. Brooks的思考

（美）Frederick P. Brooks, Jr. 著　InfoQ中文站 王海鹏 高博 译

The Design of Design

Essays from a Computer Scientist



机械工业出版社
China Machine Press

设计原本

计算机科学巨匠Frederick P. Brooks的思考

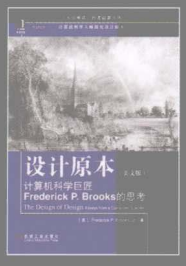
The Design of Design

Essays from a Computer Scientist

无论是软件开发、工程还是建筑，有效的设计都是工作的核心。本书将对设计过程进行深入分析，揭示有效和优雅设计的方法。

本书包含了多个行业设计者的特别领悟。作者精确发现了所有设计项目中内在的不变因素，揭示了优秀设计的过程和模式。通过与几十位优秀设计者的对话，以及他自己在几个设计领域的经验，作者指出，大胆的设计决定会产生更好的结果。

作者追踪了设计过程的演进，探讨了协作和分布式设计，阐明了哪些条件造就了真正卓越的设计者。他解释了设计过程的具体细节，包括多种预算约束条件、美学考虑、设计经验主义及工具。同时，他将这些讨论与现实中的案例结合起来，这些案例从房屋建造到IBM的Operating System/360。贯穿全书的成功的关键因素，是每个设计者、设计项目经理和设计研究者都应该知道的。



书号：978-7-111-32503-1
定价：69.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

PEARSON
www.pearsonhighered.com

上架指导：计算机/软件工程
ISBN 978-7-111-32557-4
9 787111 325574
定价：55.00元

计 算 机 科 学 丛 书

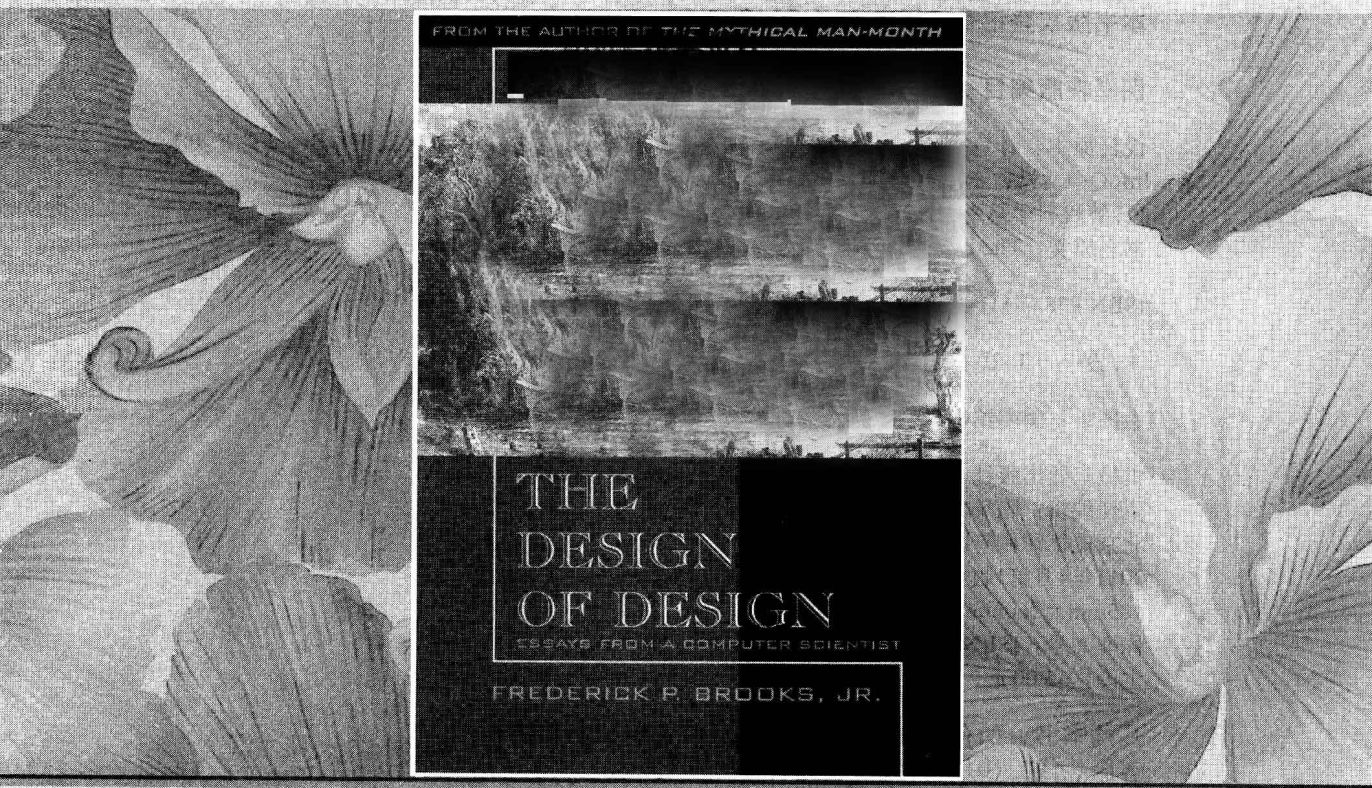
设计原本

计算机科学巨匠Frederick P. Brooks的思考

(美) Frederick P. Brooks, Jr. 著　InfoQ中文站 王海鹏 高博 译

The Design of Design

Essays from a Computer Scientist



机械工业出版社
China Machine Press

这是一部在研究和教学中将设计领域探索心得和实践经验切磋琢磨、去伪存真、取其精华的反思之作。本书几乎涵盖了有关设计的所有议题：从设计哲学到设计实践，从设计过程到设计灵感，既强调了设计思想的重要性，又对沟通中的种种细节做了细致入微的描述，以及因地制宜做出妥协的具体准则等。特别深入分析了设计模型背后的工程思想，这对设计界的研究者和实践者而言无疑具有方向性的指导意义。

本书运用大量图表和案例，深入浅出地表达了复杂艰涩的设计思想，意图刺激设计者和设计项目经理，令其深入思考设计的过程，特别是设计复杂系统的过程。本书适合各类软硬件设计者、设计项目经理、设计研究人员等。

本书版权登记号：图字：01-2010-2468

图书在版编目（CIP）数据

设计原本：计算机科学巨匠Frederick P. Brooks的思考/（美）布鲁克斯（Brooks, F. P.）著；InfoQ中文站，王海鹏，高博译. —北京：机械工业出版社，2011.1
（计算机科学丛书）
书名原文：The Design of Design: Essays from a Computer Scientist

ISBN 978-7-111-32557-4

Ⅰ.设… Ⅱ.①布… ②I… ③王… ④高… Ⅲ.软件设计 Ⅳ.TP311.5

中国版本图书馆CIP数据核字（2010）第228594号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：陈冀康 秦 健
北京瑞德印刷有限公司印刷
2011年4月第1版第2次印刷
185mm×260mm · 18.5印张
标准书号：ISBN 978-7-111-32557-4
定价：55.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反

馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育
华章科技图书出版中心

Frederick P. Brooks, Jr. 论设计原本[⊖]

很少有人像Frederick P. Brooks, Jr. 一样对软件开发的实践（而不是学术理论）产生那么大的影响。他在《人月神话》中记录了他作为IBM System/360计算机以及Operating System/360项目领导者的经验，这些经验不断地促使我们形成项目管理的观念。很难找到比他的“没有银弹：软件工程中的根本问题和次要问题”（《Information Processing 1986, Proceedings of the IFIPS Tenth World Computer Conference》, H.-J. Kugler编辑。Amsterdam: Elsevier Science, 1069-1076）一文被引用更多的文章。“银弹”的概念代表了对困难的软件和编程问题的某种近乎神奇的解决方案，这是我们的词汇表中不可缺少的一个词。

Brooks的这本新书拓展了他以前的思想，添加了对设计的本质和重要性的新领悟。毫无疑问，这本书将会成为所有专业开发者书架上必备的书籍。

这本书的副书名是“计算机科学巨匠Frederick P. Brooks的思考”。书中包含了可以作为独立的文章进行阅读的20章，每一章与其他章的耦合都比较松散。这是好事，因为大多数读者都希望阅读本书的每一章（不一定要按顺序），然后在继续下一章阅读之前思考一下所讲的内容。除了这些文章外，这本书还提供了8个案例，涉及的范围从海滩小屋的设计到IBM System/360的架构。这些案例阐释了本书中的一些重要概念。

什么是设计？Dorothy Sayers（英国作家和戏剧家，Brooks引用了他的话）说设计有3个阶段：概念构造的形成，在真实媒质上的实现，与真正用户的交互。Brooks在他的“银弹”一文中指出，第一个阶段（即概念构造）是软件工程最困难的阶段。但在1986年时，“概念构造”似乎更侧重计算机在执行指令时内部发生了什么。在这本新书中，关注的重点更多地转移到架构、外观，以及程序工件与环境的交互上。有趣的是，Brooks在第1章的开始引用了培根的话：

（新思想来自于）将一门艺术中的领悟联系并应用到另一门艺术中，历经若干次这样的经历而有所悟，脑海里自然就孕育出了“新思想”。

这清晰地表达了跨学科的灵感和锻炼。Brooks没有深入这个概念，即使在后面的内容中讨论如何“创造”了不起的设计师时，他也没这样做。

有几章讨论了设计和设计过程的模型。Brooks在这里严厉批评了流行的“理性主义者”设计模型（读者可能最熟悉的就是“瀑布方法”），并断言“理性模型过于简单”。书中指出了理性模型的诸多缺陷，包括“对理性模型的最具破坏性的批评可能是，最有经验的设计完全不是这样工作的”。（在本书中提到David Parnas多次，但没有提到他的著名文章“The Rational

⊖ 原文参见：http://www.infoq.com/articles/brooks-design-book-review。

Design Process: How and why to fake it”，这篇文章也对理性模型提出了严厉批评。）本书对其他一些设计模型也进行了讨论，包括：

- 迭代演进式开发，与此最接近的是Brooks对敏捷方法的讨论。
- Boehm的螺旋模型。
- 开源的、Raymond的“集市模型”。

这些讨论的结论是：设计需要某种过程以及该过程的模型（主要是为了沟通并支持协作），Barry Boehm的螺旋模型是Brooks认为最有希望的模型。

本书中的其他章节关注了以下几个问题：

- 协作与团队设计，包括远程协作所引发的问题。
- 关于“理性主义”与“经验主义”的讨论，Brooks自认是一个经验主义者。
- 约束条件的价值。其中印证了许多“设计”文献，这些文献来自工业、产品和图形设计师。设计草案（用于启动设计过程的文档）必须足够模糊，允许自由思考和表达，但必须清楚定义所有的约束条件。约束条件勾画出边界，创造性的、有想像力的、创新的设计将在这个边界之内发生。
- 讨论美与风格。
- 一些关于设计技术和实践的讨论，Brooks发现它们是有价值的。
- 需求、罪过与合约。
- 一个案例，讲述了为什么任务控制语言（Job Control Language, JCL，如果你没有在大主机上工作的愉快经历的话）可能是“有史以来最糟糕的编程语言”！

本书末尾用两章的篇幅讨论卓越的设计和卓越的设计师的问题。Brooks果断指出，卓越的设计来自卓越的设计师，而非卓越的设计过程。Brooks说，我们教育和培训软件专业人员的方式无助于培养出卓越的设计师。他指出，培养卓越的设计师的一个主要障碍就是缺少持续的实践和批评。

InfoQ有机会对Brooks提出了一些跟进问题，本文包括了这次访谈的内容。问题与回答如下：

开发项目有一个生命周期，要么是经典的线性瀑布式，要么是迭代增量式（敏捷）。设计是在生命周期特定阶段发生的事情，还是分布在所有阶段？

它集中发生在迭代开发的前面几次循环，但有时候发生在所有迭代中。

如果设计发生在所有的开发阶段中，是否在每个阶段中具有不同的形式、实质或要点？

当然是这样的！在第一次迭代中，总体架构是中心问题。在接下来的迭代中，设计工作集中于更精细的层面，除非是在满足最后期限之后的回溯，或者人们意识到需求的改变或新的机会。

约束条件在设计过程中扮演什么角色？（本问题的背景知识：其他领域的设计师常常依赖一份“草案”，他们预期/需要草案中有模糊之处，以便能够自由地“设计”，但他们需要清楚表达约束条件，这些约束条件定义了目标区域，最优设计只能在这个区域中产生。）

它们既促成了整体架构，又在较小程度上促成了细节设计。

是否有明显可以确定的设计“错误”，它们是否能在犯下时就可以意识到？（或者，这样

的错误是否像代码中的缺陷，通常需要在犯下之后许久才发现？）

大错是在开始时犯下的，而且很少意识到。如果最终被发现，通常是在现场实施之后。较小的错误是在编码开始时出现，或者是在第一个真正的用户测试原型时被发现。

大多数设计师认为他们的活动是高度协作的，至少是客户与设计师之间的协作，但设计更多时候涉及一个团队。您是否同意设计是一种协作？如果是这样，设计团队的地理分布或临时分布会带来什么影响？（显然，在今天的离岸外包环境中寻求并行设计，以应对软件开发的一般挑战。）

我用了两章讨论协作和远程协作。是的，今天大多数的设计都涉及团队。通用产品的设计不涉及与客户的协作，如iPad。对于为一个客户设计定制的产品，我非常喜欢对原型和代用品（如建筑的虚拟环境模型）尽早地、激烈地、频繁地、不断地进行用户测试。但是，我不认为这是与用户和客户进行协作设计。

您是否有一份清单，例如6点建议，可以总结您的书向设计者提供的最重要的经验？

1）专心研究以前设计者的工作，看看他们如何解决问题。
2）尝试弄明白他们为什么做出那样的设计决定，这是对你自己最有启发性的问题。
3）仔细研究以前设计者的风格。最好的方式是尝试用他们的一些风格勾画设计草图。
4）保存一本“草图本”，将您的想法、设计和局部设计记录下来，不论使用何种媒质。
5）在开始设计时，写下您对用户和使用方式的假定。
6）设计、设计、设计！

在您的“银弹”文章中，您谈到了“概念构建”和人类在头脑中完成这项工作时遇到的巨大困难。您觉得在这本书中一些思想是否关注并解决了概念构造这一基本困难？

肯定关注了，肯定没解决。

在第3章中，您漂亮地批评了理性设计过程，特别是瀑布式方法所包含的理性设计思想，并指出这种思想是有害的，必须抛弃。您对这种有害模型的持续存在有何看法？是否人们就是很倔强？或者尽管开发者更了解，但管理层会犯错？是否有一些文化上的偏见（尤其是西方文化上的偏见）阻碍人们抛弃瀑布模型？

第4章讨论了软件工程中的瀑布模型的持续存在（在其他学科中并不常见）是因为设计者过早期望得到有约束力的合同和确定的需求。

在第20章中，您批评了我们的教育系统（温和，但确是事实），并建议设计者需要参与“批评性实践”。Richard Gabriel长期以来一直主张计算机科学/软件工程大纲应该采用他在取得诗歌硕士学位时一样的大纲（他在很久之前获得了计算机科学博士学位）：大量的练习（每天至少一首诗）、大量的批评（来自同学和导师）、勤奋学习大师和大师的诗歌、不断自省、周期性的反省。这似乎与您的建议相似，那么您是否主张大学提供一个“软件好艺术硕士”学位？

不。熊恩在《the Education of the Reflective Practitioner》一书中提出了同样的建议，还有例子，更适合设计。所有的工程学大纲都应该强调这种方式。

哪些其他领域的研究将有助于毕业生变成卓越的软件设计师？

1）算法和数据结构是最重要的基础课程。

2）计算机硬件架构。

3）应用领域，特别是商业数据处理、数据库技术和数据挖掘。

4）心理学，特别是知觉心理学，因为用户是最重要的。

理性设计过程，实际上是所有计算机科学和软件工程，有意识地采用了宇宙的基本模型(20世纪的物理学)，即确定性的、机械的、理性的宇宙。如果那就是自然或现实，那么理性的、搜索树式的设计过程模型就很有意义。如果宇宙实际上是复杂的适应性系统，那么理性模型就会失效。您是否走得更远，奠定了复杂系统的设计或修改过程的基础，如文化或商务企业？

我不会自大到建议这样的东西。

IDEO是一家非常有名的设计公司，它的总裁Tom Kelly写了一些关于设计、设计过程和设计思考的书。他最好的一本书是《Ten Faces of Innovation》，其中他确定了每个设计团队需要的10个角色（人类学家、实验员、嫁接能手、跨栏运动员、合作者、指导者、用户体验设计师、布景师、专业护理人员和讲故事的人）。如果您知道这本书，是否类似的角色应该出现在软件设计团队中？怎样做到？

我不了解这本书。听起来有趣，而且有用。

您提到了Christopher Alexander和他的影响并在注释中提到了《The Synthesis of Form and A Pattern Language》一书。那代表了“理性的Alexander”，但“Timeless Way of Building”和他的杰作“Opus, Nature of Order”却更为“神秘化”。“神秘的Alexander”是否对您的设计和设计过程思想有所影响？

我受到了《The Timeless Way of Building》一书的影响。

软件领域的设计将大多数注意力放在人工制品上，即执行程序的计算机。越来越多的实践（但学术界还没开始）开始关注“用户体验设计”，即计算机所处的系统和生态环境。对于用户体验设计者如何从本书中获益，您是否有一些建议？

我觉得前面给出的建议同样适用于用户体验设计，但这个领域与软件工程领域相比，自由式教育更为重要。

您能列举出值得所有人学习的3名设计大师（任何领域）和3名软件设计大师，4至5个设计杰作，并简单说明为什么吗？

- 巴赫，作曲家
- 伦勃朗，画家
- Seymour Cray，超级计算机设计师
- Christopher Wren，建筑，特别是他在伦敦设计的教堂
- Nicholas Wirth，计算机语言
- Donald Knuth，算法

我不认为所有的人都应该学习同样的例子。人们需要接受范例的设计原则方面的基本教育，这样才能深入研究一个范例，欣赏这些设计问题和解决方案。但是，即使是门外汉也能欣赏一致性和设计概念的统一性。

译者序

……《诗》有之："'高山仰止，景行行止'。虽不能至，然心向往之。"

——司马迁（西汉），《史记·孔子世家》

大师之作不仅仅是指一部出自大师笔下的著作，更是特指大师的心血凝结之作。Frederick P. Brooks, Jr.，美国"两院"院士、A.M. 图灵奖和IEEE先驱奖获得者[1]、软件工程界至今脍炙人口的奠基之作《人月神话》的作者，这位令人高山仰止的大师，在创作了《人月神话》35年之后，才在2010年初推出了本书。如果说《人月神话》是Brooks刚刚完成若干个改变了全球计算系统格局的重大项目，在人生和事业的巅峰时期的激情之作，那么本书则是Brooks功成名就之后，在研究和教学中将先前在设计领域中的探索心得和实践经验切磋琢磨、去伪存真、取其精华的反思之作。可以说，比起锐气有余的《人月神话》，本书更多了几分高屋建瓴的大局观以及数十年如一日积淀而成的丰富材料，是设计领域真正的大师之作。

本书几乎涵盖了所有有关设计的议题：从设计哲学到设计实践，从设计过程到设计灵感，既强调了设计思想的重要性（第8章），又对沟通中的种种细节做了细致入微的描述（第6、7章），并且也谈到了因地制宜做出妥协的具体准则（第9、10、11章）。其中，Brooks特别强调的是设计的概念完整性（第6章），这不仅对于设计过程中步骤流转时的信息传递十分关键，并且也是沟通中最需要重点注意的地方。使用同一个术语表达不同的概念，或使用不同的术语表达同一个概念，都会给设计带来剧增的成本，甚至灾难性的后果。这一点是贯穿始终的主线之一。另一条主线就是Brooks对于理性模型的批判（第2、3、5章）。由于在现行的软件工程著作和研究论文中，对理性模型导致的直接模型——瀑布模型的推崇可谓甚嚣尘上。Brooks在此处着了重墨，深入分析了理性模型的工程师心理学渊源，解释了它何以根深蒂固，然后剖析了它的实质——以拓展设计树的方式来暴力遍历问题的解空间，最后对它的种种不足提出了针对性的批评，并指出在哪些受限的条件下方可运用理性模型（瀑布模型），而在其他场合中有哪些更好的设计模型，尤其是Boehm提出的螺旋模型。这些对于设计模型的长篇论述，特别是对其背后的工程思想的深入分析，无疑将对设计界的研究者和实践者起到方向性的指导意义。

全书的案例研究是另一大亮点，这不仅包括专门的案例章节（第21～27章），而且在进行抽象的模型和思想论述时，Brooks也时时注意运用图表和案例说话，深入浅出地表达复杂艰涩的思想。并通过层次丰富的案例，展示了设计既能治大国，又可烹小鲜的强大力量和无穷魅力。比如，为了论述抵制需求蠕变的必要性时，他首先以和美国军方要员的一段对话切入话题（第4章），给读者以直观且深刻的印象。这不仅表明了即使在军事领域，设计的准则和影响仍然适用，也不经

意间揭示了作者和美国国防部——全世界最尖端的科研和工程的研发和实践基地——的深入合作关系。Brooks以揶揄的方式对待这些大企业的高管们的案例并非仅此一隅，在讲述僵死、拙劣的规格如何造成最终产品的可笑妥协时，他又举一例，美国联邦航空局的一个令人匪夷所思的系统规格造成了在最终产品中不得不以禁用一个完整系统的部分功能的方式“凑合”成一个虽然符合规格却造成不小浪费的系统，而这些最终都是由纳税人买单（第11章）。要知道，这些都是动辄涉及上百亿美元的大项目，Brooks在其中的谈笑风生绝对是一种举重若轻的大将风度。可是另一方面，Brooks又会在讲述设计中的约束时，在多处提到对自家房屋进行建筑设计和拟订整修方案时遇到的各种困难，并一一指出如何应对：有些约束只要改变一下思路就会消失，有些约束虽然无可避免但可以最小化，有些约束反映了原始设计方案中的方向性错误，等等（第1、3、17、18章）。这种十分贴近读者生活实际的例子不仅一下拉近了作者和读者的思想距离，同时也更说明Brooks热爱设计工作到何种程度，连一般人视作生活小节之处也不愿意放过，而是把它作为设计工程来研究一番[2]。顺便说一句，Brooks在建筑设计方面也决非业余，他曾经参与他工作至今的北卡罗来纳大学的西特森厅设计，是设计委员会的正式成员[3]，这在本书中也有提及（第4章）。

当然，Brooks毕竟是软件工程业界的先驱，正如在他的《人月神话》或任何一本主要文献中一样，我们都能够在他的作品的字里行间感受到计算机体系结构刚刚诞生的黄金年代充满了怎样的设计思想和工程实践的生机和活力。而Brooks也十分擅长专业史料的记载和整理，并且以他独有的方式为读者展示出极为清晰的脉络。比如，他主持设计的System/360系统不仅是当代操作系统在实际意义上的先祖和典范，而且它本身仍然活跃在历史舞台上：Brooks在书中指出，System/360和OS/360上的应用程序至今仍然可以完全兼容地运行在当今的后续体系结构之上，包括晚至2007年发布的64位IBM Z/90机型——一种System/360体系结构的直接后裔机型（第24章）。正是通过这样上承开天辟地之洪荒巨擘，下接耳熟能详之主流系统和应用的史诗式描述，让我们在充分领略软硬件发展史的无限风光的同时，也深切地感受到用心设计会带来数十年前后一贯的、可持续发展的产品，而这些产品及其反过来促进的设计思想和方法论又怎样彻底地改变了我们每一个人生活、工作和沟通的方式。这不也正是包括我们自己在内的一代代设计师和架构师投身于此的原动力吗？

本书由我、王海鹏与InfoQ中文站的张龙、黄璜共同翻译完成。虽然与几位的合作还是第一次，但是整个过程进行得非常顺利和愉快。在全书的翻译实践中，我本人收获极大。能够逐字逐句地研读本书，已经是充分的精神享受——Brooks的文字无疑是值得一读再读的。再经过整个团队的协作和努力，把它的内容和意义带给中国的千百万读者，这对我们翻译工作者来说，已经是无上的嘉赏和成就了。当然，由于我们的水平所限，缺点和错误在所难免，希望广大读者不吝指正，以便在再版时予以修订。

在本书的翻译过程中，机械工业出版社和InfoQ中文站给予了我们很大的支持、鼓励和帮助。UMLChina于2010年6月下旬举办了一场“Brooks新作暨《人月神话》35周年讨论会”[4]，在本书出版前提供了一个与读者交流和讨论的机会。本书译稿成稿之前，多位友人阅读了翻译初稿并给予本人许多可贵的修正意见，尤其是和我一起工作的张龙、黄璜、王海鹏，盛大创新院的郭忠祥院长和刘海平工程师，SAP中国的范德成工程师，MBK Partners私募基金公司的章子琦分析师，富士康中国研发集团的Carl Giardina总监，以及微软总部的米琦工程师，在此一并致谢。当然，家父高学栋博士也通读了全稿并给予了我不少有关文法和表达的中肯意见。我也想借此机会向在工作和生活中给了我莫大支持的父母和家人表达我内心最深处的敬意和谢

意，希望本书的出版能给他们带去快乐。

高博

2010年11月

于盛大创新院上海总部

译者简介

高博 2004年毕业于上海交通大学计算机系，在微软公司和惠普公司有多年项目和管理经验。对程序设计语言、云计算、软件测试方法学、软件架构设计和软件项目管理方向有浓厚兴趣。近年来翻译出版了《C++：99个常见编程错误》、《微软的软件测试之道》等多本书籍。联系方式：feedback@gaobo.org，订阅博客：feed.gaobo.org。

王海鹏 1994年毕业于华东师范大学。拥有双学士学位。咨询顾问、培训讲师、译者和软件开发者。已翻译20本软件开发书籍，主题涵盖敏捷方法学、需求工程、UML建模和测试。拥有15年软件开发经验，目前主要研究领域是软件架构和方法学，致力于提高软件开发的品质和效率。

张龙 同济大学软件工程硕士，InfoQ中文站Java社区编辑，满江红开放技术研究组织成员。热衷于编程，对新技术有强烈的探索欲，对Java轻量级框架有一定研究。目前对Mac、iPhone/iPad、Android开发、动态语言及算法具有浓厚兴趣。翻译出版了《Dojo构建Ajax应用程序》、《Spring高级程序设计》等书籍。拥有5年的Java EE培训讲师经验。联系方式：zhanglong217@yahoo.com.cn，博客地址：http://blog.csdn.net/ricohzhanglong，新浪微博：http://t.sina.com.cn/fengzhongye，欢迎follow。

黄璜 2007年毕业于重庆邮电大学。曾在某外包公司工作3年，目前在一家初创公司做开发工作。担任InfoQ中文站社区编辑两年，对于业界新动向和新技术有强烈的兴趣。目前专注于网站架构、分布式以及算法。联系方式：alexhuang1984@gmail.com。

注释

1. http://www.cs.unc.edu/~brooks/FPB_BIO.CV.04.2007.pdf，最近访问于2010年11月。

2. http://www.cs.unc.edu/~brooks/DesignofDesign/kitchen_design_notes.pdf，这份扫描件展示了作者把每一个建筑细节都当做设计问题来研究的严谨工作和生活态度。http://www.cs.unc.edu/~brooks/DesignofDesign/house/start_here.html，他甚至还把详细的设计树数字化并放入网站供读者了解每一个设计细节以及它们在决策中的定位，这种一丝不苟的工作习惯着实令人感动。最近访问于2010年11月。

3. http://www.cs.unc.edu/~brooks/DesignofDesign/experiences.html，“Buildings”部分，最近访问于2010年11月。

4. http://www.umlchina.com/Chat/mmm100620.htm，包括讨论会幻灯片与对话的音频与文字记录（会议由王海鹏主持），最近访问于2010年11月。

前 言

我写这本书是为了刺激设计者和设计项目经理，让他们深入思考设计的过程，特别是设计复杂系统的过程。本书从工程师的视角关注实用性与有效性，同时也关注效率和优雅性[1]。

谁应该阅读这本书

《人月神话》的目标读者是“职业程序员、职业经理，特别是管理程序员的职业经理”。在那本书中，我讨论了团队开发软件时，实现概念完整性的必要性、困难和方法。

本书在相当大的程度上扩大了范围，并添加了我35年来学到的经验。设计经验让我确信，各种不同设计领域的设计过程包含一些不变的因素。因此本书的目标读者是：

1）**各类设计者**。排除直觉的系统化设计将得到普普通通的跟随式产品和仿冒产品，没有系统的直觉设计将得到充满缺陷的、不切实际的产品。如何将直觉和系统化方法融合在一起？如何成长为一名设计师？如何在一个设计团队中发挥作用？

虽然我针对非常多的系统进行了论述，但我期望读者是偏重于计算机软件和硬件的设计者，面对这样的读者我可以提供具体的阐述。因此我在这些领域的某些例子中会涉及技术细节。其他读者可以跳过这些细节。

2）**设计项目经理**。要避免灾难，项目经理在设计他的设计过程时，必须融合理论与口口相传的经验，而不是仅仅照搬某种过于简化的学术模型，也不能临时设定一个过程，而不参考他人的理论或经验。

3）**设计研究人员**。对设计过程的研究已经成熟，这是好事，但并不是一切都好。已发表的研究成果越来越关注更狭窄的主题，大问题讨论得越来越少。对精确的期望和对“设计科学”的期望可能使得科学研究之外的出版物受阻。我建议设计思考者和研究者重新关注这些大问题，即便是在社会科学方法没有太大帮助的时候。我相信他们也会思考我的论述是否具有通用性，我的观点是否正确。我希望为他们的学科领域提供服务，将他们的一些研究结果带给实践者。

为什么要再写一本关于设计的书

创造东西是一种快乐，是一种极大的满足。J. R. R. Tolkien说上帝给了我们发明创造的能力，作为一件礼物，纯粹是为了让我们快乐[2]。毕竟，“千山上的牲畜也是我的。……我就算饿了，也不用告诉你。[3]”设计本身就是快乐的。

很多设计者从心理上和实践上都没有对设计过程进行很好的理解。这不是因为缺少研究。

许多设计者反思了他们自己的设计过程。研究的动机之一就是，在所有的设计领域，最佳实践和平均实践之间存在着巨大的鸿沟，平均实践和半吊子实践之间也是如此。大部分的设计成本是返工，即纠正错误，这通常达到总成本的1/3。平庸的设计肯定是浪费了世界的资源，破坏了环境，影响了国际竞争力。设计很重要，设计教育也很重要。

所以，根据推理，系统化设计过程将提升平均实践的水平，而结果也确实如此。德国的机械工程设计者们显然是首先采用了这一规划[4]。

随着计算机和之后人工智能（AI）的出现，设计过程的研究受到了极大的刺激。最初人们希望，AI技术不仅能够在过去人类主宰的领域中承担许多例行设计的工作，甚至能够产生杰出的设计[5]。这种希望迟迟没有实现，而我本人觉得不可能实现。设计研究形成了一门学科，有一些专门的学术会议、期刊和许多研究项目。

既然已经有了这么多认真的研究和系统的处理，为什么还要再写一本书？

首先，设计过程自第二次世界大战以来，有了非常大的变化，而人们很少讨论这些变化。对于复杂产品的设计，团队设计越来越成为常态。团队常常在地理上是分散的。设计者越来越脱离产品的使用和实现，通常他们不再亲手打造他们设计的东西。各类设计者现在都陷在计算机模型中，而不是陷在图纸中。正式设计过程的教育越来越广泛，而且通常是雇主强制要求的。

其次，仍然存在许多误区。当我们试图教学生怎样做好设计时，我们在理解上的差异就变得很明显了。Nigel Cross是设计研究领域的一位先行者，他追踪了设计过程研究变化的4个阶段。

1）规定（prescription）一个理想的设计过程

2）描述（description）设计问题的内在本质

3）观察（observation）设计活动的现实

4）反思（reflection）设计的基本概念[6]

我在我人生的60年时间里涉及了5种设计领域：计算机架构、软件、房屋、图书和组织机构。在每个领域，我都承担过团队中主设计者和协作者的角色[7]。我对设计过程的兴趣由来已久，我在1956年的论文是“The analytic design of automatic data processing systems（自动化数据处理系统的解析设计）[8]”。也许现在是时候进行成熟的反思了。

这是一本怎样的书

令我非常吃惊的是，以下这些过程极为类似！思维的过程、人与人的交互、迭代、约束条件和劳动，都有很大的相似性。本书中反思的东西可能是隐藏在这些设计活动背后不变的设计过程。

虽然计算机架构、软件架构的历史不长，对它们的设计过程的反思也不多，但建筑设计和机械设计已经有很长的历史和荣耀的过去。在这些领域，设计理论和设计理论家都很多。

我是一名职业设计师，我所工作的领域中对设计的反思还不多，而在那些得到长期深入反思的领域，我是一名业余设计师。所以我将尝试从历史较长的设计理论中提取一些经验，应用于计算机和软件的设计。

我相信“设计科学”是一个不可能完成的目标，实际上也是一个具有误导性的目标。这种解放思想的怀疑论让我们能够从直觉和经验的角度进行探讨，包括其他设计者的经验，他们很客气地和我分享了他们的领悟[9]。

所以我提供的既不是一本教科书，也不是一本包括一致论证的专著，而是一些观点文章。虽然我试图补充一些有用的参考和注解，探索一些隐秘的小路，但我仍建议读者先从头到尾阅读每篇文章，忽略这些注解和参考，然后再回过头来探索这些小路。所以我将它们藏在每章的末尾。

某些案例研究提供了一些具体的例子，文章中参考了这些例子。选择这些例子并不是因为它们很重要，而是因为它们体现出了某种经验，我基于这些经验得出了结论和观点。我特别喜欢关于房屋功能设计的那些经验，任何领域的设计者都可以参考它们。

作为主设计师，我完成了3所房屋的功能设计（详细的平面图设计、照明、电气和管道）。将房屋功能设计过程与复杂计算机硬件和软件的设计过程进行比较和对比，这帮助我提出了设计过程的“精髓”，所以我用它们作为我的案例，并且相当详细地介绍了这些过程。

通过反思发现，许多案例研究具有惊人的共同特点：**最大胆的设计决定，不论是谁做出的，都为优秀的结果作出了巨大的贡献**。这些大胆的决定有时是因为远见，有时是因为绝望。它们总是在赌博，要求额外的投入，以期得到好得多的结果。

致谢

本书的书名借鉴自40多年前Gordon Glegg的一部著作，他是一位富于创新精神的机械设计师、一个有风度的人、一位有吸引力的剑桥讲师。我很荣幸地在1975年与他共进午餐，感受到了他对设计的热情。他的书名准确地概括了我的尝试，所以我怀着感激与尊敬之情，沿用了这一书名。[10]

我感谢Ivan Sutherland对我的鼓励，他在1997年建议将一本讲义发展成一本书，并在10多年后对草稿提出了尖锐的批评，促进了巨大的改进。这使我后来的智力发展历程受益良多。

如果没有北卡罗来纳大学教堂山分校资助的3个研究项目以及系主任Stephen Weiss和Jan Prins的支持，这本书是不可能完成的。我在剑桥大学受到了Peter Robinson的亲切接待，在伦敦大学学院受到了Mel Slater以及他们的系主任和同事的亲切接待。

美国国家科学基金会（NSF）的计算机与信息科学工程（CISE）理事会的设计科学（Science of Design）项目由助理理事长Peter A. Freeman发起，该项目为本书的完成和相关网站的准备提供了最有用的资助。这笔资助让我能够访谈许多设计者，并能够在过去几年里将主要工作集中在这些文章上。

我非常感谢许多真正的设计师，他们与我分享了他们的领悟。我用一张致谢表列出了受访者，并在末尾列出了评阅者。有几本书提供了特别多的信息，对我产生了很大的影响，我在第28章中列出了这些书。

我的妻子Nancy是其中一些工作的共同设计者，她一直为我提供支持和鼓励，我的孩子Kenneth P. Brooks、Roger E. Brooks和Barbara B. La Dine也是一样。Roger对手稿进行了仔细的复查，并对每章内容提出了几十项建议，从概念到标点符号都有。

我要感谢在北卡罗来纳大学获得的强大后勤支持，这种支持来自Timothy Quigg、Whitney Vaughan、Darlene Freedman、Audrey Rabelais和David Lines。Peter Gordon是Addison-Wesley的出版合作者，他提供了特别的鼓励。Julie Nahil是Addison-Wesley的全职产品经理，Barbara Wood是文字编辑，他们提供了无比专业的技能，付出了特别的耐心。

John H. Van Vleck是诺贝尔物理奖获得者，当我在哈佛大学工程和应用学院Aiken的实验室读研究生时，他是那儿的院长。Van Vleck非常注重工程实践要建立在牢固的科学基础之上。他领导了美国工程教育从设计向应用科学的转变，这种转变富有朝气。就像钟摆摆到极点，反作用就会出现，教授设计从那时起一直引起争论。我非常感谢我在哈佛的3位老师，他们从未丧失对设计重要性的深刻理解并教授设计。他们是Philippe E. Le Corbeiller、Harry R. Mimno和我的导师Howard H. Aiken。

注释

1. 本书的封面图片是基于Smethurst (1967)《The Pictorial History of Salisbury Cathedral》，他说："……除了圣保罗大教堂之外，索尔兹伯里（Salisbury）大教堂是仅有的一座英国教堂，其所有内部结构都是按照一个人（或一个两人团队）的设计建造的，并且没有中断地完成。"

2. Tolkien (1964)，《Tree and Leaf》。

3. 诗篇50:10,12。强调是添加的。

4. Pahl和Beitz (1984)，在1.2.2节中追踪了这段始于1928年的历史。他们自己的书《Konstructionslehre》历经7版，可能是最重要的系统化总结。我在所有领域中区分设计过程的研究和设计规则的研究。这些历史已经很悠久了。

5. 主要的专论是Herbert Simon的《The Sciences of the Artificial》(1969, 1981, 1996)，非常有影响力。

6. Cross (1983)，《Developments in Design Methodology》。

7. 具体的设计经历表包含在网站的附加材料中：http://www.cs.unc.edu/~brooks/DesignofDesign。

8. Brooks (1956)，"The analytic design of automatic data processing systems，"哈佛大学博士论文。

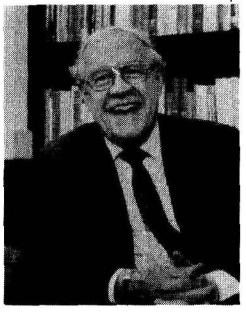
9. 我因此没有对下面网站中所说的设计方法学目标作出贡献： http://en.wikipedia.org/wiki/Design_methods（我在2010年1月5日访问了这一网址的内容）。

我们的挑战是将个人的经验、框架和观点转变成共享的、能理解的知识，最重要的是，转变成可传递的知识。Victor Margolin指出了3点困难，（其中一点是）："……个人设计经验的著作太关注个人叙述，导致了个人的观点，而不是关键的群体共享观点。"

对此我必须承认，"你说对了"。

10. Glegg (1969)，《The Design of Design》。

作者简介



Frederick P. Brooks, Jr. 是北卡罗来纳大学计算机科学系的Kenan教授。他因担任IBM System/360系统开发的项目经理而以“IBM System/360之父”而闻名于世，后来又担任了Operating System/360软件项目设计阶段的项目经理。因为这些工作，他、Bob Evans和Erich Bloch荣获了1985年的美国国家技术奖。在此之前，他曾是IBM Stretch和Harvest计算机的架构师。

在北卡罗来纳大学教堂山分校，Brooks博士创建了计算机科学系，并从1964年～1984年担任该系的系主任。他曾在美国国家科学委员会和防御科学委员会工作。他目前的教学和研究工作集中于计算机架构、交互式计算机图形和虚拟环境。

目 录

出版者的话
Frederick P. Brooks, Jr. 论设计原本
译者序
前言
作者简介

第一部分 设计之模型

第 1 章 设计之命题 ……3
培根所言是否正确 ……3
什么是设计 ……3
何为真实？设计的概念 ……4
对于设计过程的思考 ……6
设计类别 ……7
注释 ……7

第 2 章 工程师怎样进行设计思维——理性模型 ……11
模型概览 ……11
该模型的构思从何而来 ……12
理性模型有哪些长处 ……13
注释 ……14

第 3 章 理性模型有哪些缺陷 ……17
我们在初始阶段并不真正地知道目标是什么 ……17
我们通常不知晓设计树的样子——一边设计一边探索 ……18
（设计树上的）节点实际上不是设计决策，而是设计暂定方案 ……19
有用性函数无法以增量方式求值 ……19
必要条件及其权重在持续变化 ……20
约束在持续变化 ……21
对于理性模型的其他批评 ……23

但是，尽管有这些缺陷和批评，理性模型仍然不屈不挠地存在 ……24
那又如何？我们的设计过程模型真的那么事关紧要吗 ……24
注释 ……26

第 4 章　需求、罪念以及合同 ……29

一段恐怖往事 ……29
殊为不幸，无独有偶 ……30
抵制需求膨胀和蠕变 ……31
罪念 ……32
合同 ……32
一种合同模型 ……33
注释 ……34

第 5 章　有哪些更好的设计过程模型 ……37

为什么要有一个占主导地位的模型 ……37
共同演化模型 ……38
Raymond的集市模型 ……39
Boehm的螺旋模型 ……40
设计过程模型：第2～5章的讨论小结 ……41
注释 ……42

第二部分　协作与远程协作

第 6 章　协作设计 ……45

协作在本质上是好的吗 ……45
团队设计是现代标准 ……46
协作的成本 ……48
挑战是概念完整性 ……49
如何在团队设计中获得概念完整性 ……50
协作何时有帮助 ……51
协作何时无用——对设计本身 ……55
两人团队很神奇 ……56
对于计算机科学家意味着什么 ……57
注释 ……57

第 7 章　远程协作 ……61

为什么要远程协作 ……61
到那里，做那事——IBM System/360计算机系列的分布式开发，1961～1965 ……62
让远程协作有效 ……63

远程协作的技术 ……64
注释 ……66

第三部分 设计面面观

第 8 章 设计中的理性主义与经验主义 ……71
理性主义与经验主义 ……71
软件设计 ……72
我是个铁杆的经验主义者 ……72
其他设计领域中的理性主义、经验主义与正确性 ……73
注释 ……74
第 9 章 用户模型——错误胜过含糊 ……77
明确的用户与用例模型 ……77
团队设计 ……77
假如事实不可用该如何是好 ……78
注释 ……79
第10章 英寸、盎司、位与美元——预算资源 ……81
何谓预算资源 ……81
美元并非万灵丹 ……81
即便美元也有不同，替代品剖析 ……82
预算资源是可变的 ……82
那又如何 ……83
注释 ……84
第11章 约束是我们的朋友 ……87
约束 ……87
不完全如此 ……88
设计悖论：通用的产品要比特定用途的产品更难以设计 ……90
注释 ……92
第12章 技术设计中的美学与风格 ……95
技术设计中的美学 ……95
何谓逻辑美 ……96
技术设计中的风格 ……98
何谓风格 ……99
风格的属性 ……100
要想获得一致的风格——记录下来 ……101

如何获得良好的风格 ……101
注释 ……102

第13章 设计中的范本 ……105

很少会有全新的设计 ……105
范例的角色 ……105
计算机与软件设计呢 ……106
学习范本的设计原理 ……107
如何训练才能改进基于范本的设计 ……109
范本——懒惰、创意与自满 ……110
注释 ……111

第14章 专业设计者缘何犯错 ……115

错误 ……115
曾经最糟糕的计算机语言 ……116
JCL缘何是这样的 ……117
小结 ……118
注释 ……119

第15章 设计的分离 ……121

设计与使用和实现的分离 ……121
为什么分离 ……122
分离的结果 ……122
补救措施 ……122
注释 ……124

第16章 展现设计的演变途径和理由 ……127

简介 ……127
知识网线性化 ……128
我们的设计演变途径记录 ……128
我们研究房屋设计过程的过程 ……129
深入设计过程 ……130
决策树与设计树 ……132
模块化与紧密集成的设计 ……132
Compendium和可选工具 ……133
DRed：一个诱人的工具 ……135
注释 ……136

第四部分　一套计算机科学家进行房屋设计的梦想系统

第17章　计算机科学家的建筑设计理想系统——从思维到机器 ……139
挑战 ……139
一个设想 ……139
从思维到机器输入的设想 ……141
说明动词 ……142
说明名词 ……143
说明文字 ……144
说明助词 ……144
说明视点和视图 ……145
注释 ……147

第18章　计算机科学家的建筑设计理想系统——从机器到思维 ……149
双向通道 ……149
视觉显示——多并发窗口 ……149
声音展示 ……152
触觉展示 ……153
泛化 ……153
可行性 ……153
注释 ……153

第五部分　卓越的设计师

第19章　卓越的设计来自卓越的设计师 ……157
卓越的设计和产品过程 ……157
产品过程：优点和不足 ……158
观点碰撞：过程抑制，过程不可避免，如何处理 ……161
注释 ……162

第20章　卓越的设计师从哪里来 ……165
我们必须教他们设计 ……165
我们必须为卓越设计而招募人才 ……166
我们必须深思熟虑地培养他们 ……167
管理他们时必须发挥想像力 ……168
必须严密地保护他们 ……169
把自己培养成一名设计师 ……170
注释 ……172

第六部分 设计空间之旅：案例研究

第21章 案例研究：海滨小屋“View/360” ……177

亮点和特性 ……177
背景介绍 ……177
目标 ……178
机会 ……179
约束条件 ……179
设计决定 ……179
考虑正面 ……181
小屋的尺寸 ……182
设想的开始 ……182
在设计之后，构建之前的设计改动 ……183
在框架和外墙完成和初次入住之后的设计改动 ……183
评估（在37年后） ……184
学到的一般经验 ……187

第22章 案例研究：增加厢房 ……189

亮点和特性 ……189
背景介绍 ……189
目标 ……191
约束条件 ……192
非约束条件 ……193
事件 ……193
设计决定和迭代 ……193
评估——成功与未解决的缺点 ……198
学到的一般经验 ……199
注释 ……199

第23章 案例研究：厨房重新建模 ……201

亮点和特性 ……201
背景介绍 ……201
目标 ……202
机会 ……202
约束条件 ……203
关键宽度预算的推理 ……204
长度预算的推理 ……205

其他设计决定 ……206
评估 ……207
满足的其他迫切需求 ……207
在设计中使用图纸、CAD、模型、仿真模型和虚拟环境 ……208
学到的一般经验 ……209
注释 ……210

第24章 案例研究：System/360体系结构 ……213

亮点和特性 ……213
项目介绍和相关背景 ……213
目标 ……215
机遇（截至1961年6月） ……216
挑战和限制 ……216
最重大的设计决策 ……217
里程碑事件 ……219
结果评估 ……219
取得的经验教训 ……222
注释 ……223

第25章 案例研究：IBM Operating System/360操作系统 ……225

亮点和特性 ……225
项目介绍和相关背景 ……226
接受挑战 ……228
设计决策 ……229
结果评估 ……231
设计师团队 ……232
取得的经验教训 ……233
注释 ……233

第26章 案例研究：《Computer Architecture: Concepts and Evolution》图书设计 ……235

亮点和特性 ……235
项目介绍和相关背景 ……236
项目目标 ……236
机遇 ……237
约束 ……237
设计决策 ……237
结果评估 ……237

经验教训 ……238

第27章 案例研究：联合计算中心组织：三角区大学计算中心 ……241

亮点和特性 ……241
介绍与内容 ……242
目标 ……243
机会 ……243
约束 ……244
设计决策 ……244
董事会所考虑的投票方案 ……244
测量评估 ……245
经验总结 ……246
注释 ……246

第28章 推荐阅读 ……249

致谢 ……251

参考文献 ……255

第一部分

设计之模型

螺旋楼梯

第1章

设计之命题

（新思想来自于）将对一门艺术的领悟联系并应用到另一门艺术中，历经若干次这样的经历而有所悟，脑海里自然就孕育出了（新思想）。

——弗朗西斯·培根爵士（1605）《The Two Books of the Proficience and Advancement of Learning》卷二，第10章

很少有工程师和创作者……能够通过探讨对方的专业领域而互有所得。我的建议是，他们可以相互探讨设计……（然后）彼此分享在这种专业的创造性设计过程中的经验。

——Hebert Simon（1969）《The Sciences of the Artificial》

培根所言是否正确

弗朗西斯·培根爵士的假设正是我们所面临的挑战。设计过程本身是否存在那些适用于广泛设计载体的不变的属性？如果答案是肯定的，那么在一个设计载体中，设计人员就可能在攻克该载体特有的困难的过程中积累经验，从而具有比其他人对一些原则的更为清晰的理解。此外，某些载体比其他载体拥有更长远的设计及元设计（即设计之设计）的历史，比如建筑。如果这些都是正确的，并且培根的结论正确，那么不同载体中的设计人员就可以通过比较自己的经验与见解而在他们自己所处的技艺领域中学到新知识。

什么是设计

《牛津英文词典》对设计这个动词作了如下定义：

对……形成计划或模式，运用思维整理或考量……以便后续执行。

这一定义的精髓在于计划、思维和后续执行。所以，一个设计（名词）是一种被创造出来的事物，它先于被设计的事物出现且与之相关，但又有所区别。英国作家、戏剧家Dorothy Sayers在她那本发人深省的著作《The Mind of the Maker》里，将创作的过程分为了三个不同的阶段，她称之为构想（Idea）、精神（Energy）（或实现（Implementation））以及交互

(Interaction)。[1]这代表着：

1）概念性构想的形成；
2）在真实的媒体中实现；
3）在真实的体验中与用户交互。

在这一概念中，无论是一本书，或是一台电脑、一个程序，首先是一种概念性的构造，它独立于时间和空间，而其精髓是在作者的脑海中完成的。然后通过钢笔、墨水和纸或者硅和金属在真实的时间和空间中得以实现。当某人读到这本书、使用了这台计算机或运行了此程序时，用户与创作者的思想达到了交互，这种创作就完成了。

在我之前的一篇文章中，我将构建软件的工作分为根本的（essence）和次要的(accident)。[2]（这一亚里士多德语言并非要贬低软件构建的次要部分。在现代语言中更易理解的术语应当是essential和incidental。）我称之为根本的软件构造部分是形成其概念性结构的心智过程，称之为次要的部分是其实现过程。交互，也就是Sayers所说的第三步，发生在软件使用之时。

因此，设计就是脑力的构思，即Sayers称之为“构想”的部分。它可以在任何的实现开始之前完成。曾有一次，莫扎特的父亲询问他关于三周内要交付公爵的一部歌剧进度如何，莫扎特当时的回应既让我们感到震惊，又清晰地阐明了这一概念：

一切都谱成了，只是还没写下来而已。

——给利奥波德·莫扎特的信（1780）

对大多数的创作者来说，构思的不完整性和不一致性只有到了实现的时候才变得明显起来。因此，记录、实验和“解决”成为了理论家们的关键原则。

构想、实现和交互这三个阶段的操作是循环进行的。实现为另一轮必须完成的设计周期创造了空间。因此，莫扎特使用钢笔和纸实现了他的歌剧构思，而指挥家通过与莫扎特的作品进行交互，理解并形成了自己的演绎，又通过乐队和歌手将其实现。最终通过观众参与的交互而完成整个过程。

一个设计是一个被创造出的事物，与之相关的是一个设计过程，我将此过程称之为设计，不加任何修饰。还有一个是动词意义的设计，即进行设计。这三者是紧密相关的，我相信在具体的环境中就不会混淆它们的含义了。

何为真实？设计的概念

如果许多个体有着共同的名字，那么我们可以认为它们同样有着相应的概念或形式——明白我所说的吗？

明白。

让我们以任意一个普通的事物为例。我们的世界中有许许多多的床和桌子，是吗？
是的。
但这里仅仅存在两个它们的概念或形式：一个是床的概念，一个是桌子的概念。
确实如此。
而任何工匠都是遵循这种概念来制作我们所使用的床和桌子的。

—柏拉图（公元前360年），《理想国》第十卷

在2008年的第7届设计思想研讨会上，每个发言人都对四个同样的设计小组会议作报告。[3] 视频和打印件都提前很好地分发下去了。

来自雷丁大学的Rachael Luck在架构会谈中提出一个之前没有引起任何人注意，而后又被大家一致认同的实体：设计概念。[4]

毫无疑问，架构师和客户总是不断提到这一共享的不可见的实体。演讲者时常会对着画面作出各种含糊的手势，但显然他们并不是在指向画面的某一部分或者那其中的某一特定事物。通常，他们所关注的是开发中的设计概念的完整性。

Luck的见解让设计概念拥有了其自身的地位，这于我本人的经验有着强烈的共鸣。在开发IBM System/360大型计算机家族的单一架构的时候（1961～1963），尽管从来没有正式命名过，但这样的实体始终存在于架构小组内部。得益于Gerry Blaauw的远见卓识，我们将System/360的设计活动明确地分成了架构、履行和实现三个部分。[5] 其基本思想是整个计算机家族对程序员呈现统一的接口，即架构；而根据性能和价格的不同可以有多个并存的实现体（见第24章）。

多个实现的同时性伴随着几个工程经理的竞争，这些驱动着形成一个统一漂亮的架构，并且避免了为节约成本而作出较小的妥协。然而这种力量仅是来自于架构师们的本能和愿望，他们每个人都想做出一台漂亮的机器。[6]

随着架构设计的不断发展，我发现了一件乍一看很奇怪的现象。对于架构小组而言，真正的System/360是设计概念本身——一台柏拉图式的理想机器。那些在工程车间建造中的机械式的或电子的Model 50、Model 60、Model 70和Model 90等，只不过是模仿那台真正的System/360的柏拉图式机器的影子。真正System/360的最完整最忠实的体现，不在那些以硅、铜或者钢的形式组成的物理计算机上，而是存在于《IBM System/360操作原理》这本程序员的机器语言手册的文字和图表里。[7]

后来在View/360海滨小屋（见第21章）的建造中，我也有类似的体验。它的设计概念在构建活动开始的很早以前就已经成型。历经了许多版本的绘图与纸板模型搭建，其概念始终贯穿其中。

非常有趣的是，我从未在OS/360软件家族中感觉到这样的设计概念实体。也许它们的架构师有这种感受，又或许我对其概念框架的理解还没有到了如指掌的程度。也许设计概念没有在我这里萌发的一个原因是OS/360实际上是分别由四个部分混合而成的：一个主控制器、一个调度器、一个I/O控制器以及一个庞大的编译器和实用工具软件包（见第25章）。

价值何在

在设计对话中将不可见的设计概念转化为真正的实体是否会带来积极的价值呢？我认为是的。

首先，良好的设计具有概念性的完整性——统一、经济、清晰。它们不仅可以工作，而且能带来快乐，正如维特鲁威首次阐述的那样。[8] 我们使用诸如优雅、利落、漂亮这样的术语来形容桥梁、奏鸣曲、电路、自行车、计算机以及iPhone。辨析出设计概念这样一个实体，可以帮助我们在自己独自设计时去追寻这种完整性，有助于在团队设计时围绕这一概念一起工作，也有助于将它传授给年轻人。

其次，以这样的方式经常提及设计概念，对于一个设计团队内的沟通有极大的帮助作用。概念的统一是一种目标，它只有通过大量的对话才能达到。

就设计概念本身而言，比起由它衍生而出的表达或是部分细节，会话要直接得多，也是焦点所在。

因此，电影制片人使用故事板来保持其设计会话始终关注设计概念而不是实现细节。

关注细节，当然就会将不同版本的概念之间的冲突暴露出来，并迫使其得到解决。例如，System/360架构需要一个十进制数据类型，以桥接拥有成千上万用户的IBM十进制机器。我们所开发的架构中已经有了几个数据类型，包括32位的定点补码整数和可变长的字符串。

十进制数据类型可以被定义成类似于这两者当中的任意一个。那么哪一个才更适合System/360的设计概念呢？两方面对此都拿出了强有力的论据，这背后的力量直接依赖于个人对于设计概念不同的理解。一些架构师脑海中的设计概念反映的是早期的科学计算机，而其他架构师脑海中的概念反映的是早期的商务计算机。System/360有明确的设计目标，对于这两种应用都应提供良好的支持。

我们选择以字符串数据类型为基础来建模十进制数据类型，对于绝大部分特定的十进制数据类型用户群体，即IBM 1401的用户来说，这是最熟悉的数据类型。如果再给我一次机会，我仍会做出这样的决定。

对于设计过程的思考

关于设计的思想由来已久，至少可以追溯到Vitruvius（逝于公元前15年）。他于古典时期（classical period）写就的《De Architectura》被奉为设计的奠基之作。而随后达·芬奇的《Notebooks》（1452—1529）及Andrea Palladio的《Four Books of Architecture》（1508—1580）则可称作是这一领域里的里程碑。

而对于设计过程本身的思考则是近现代的事。Pahl和Beitz将其追溯到1852年由Redtenbacher所带来的德国思潮，这一思潮是由机械化的兴起而激发的。[9] 对于我本人而言，关键的里程碑要数Christopher Alexander的《Notes on the Synthesis of Form》（1962）、Herbert

Simon的《The Sciences of the Artificial》（1969）、Pahl和Beitz的《Konstructionslehre》（1977），以及设计研究学会（Design Research Society）的成立和《Design Studies》（1979）这本杂志的创刊。

Margolin和Buchanan所编撰的《The Idea of Design》（1995）收录了来自《Design Issues》期刊的23篇文章，大部分是关于设计评论与理论的，并“对理解设计有所影响的哲学问题作了少许探讨”（第XI页）。

我的《人月神话》（1975, 1995）反映了IBM OS/360的设计过程，它后来发展成为了MVS及其后继的产品系列。这本书着重描述该设计与开发项目中人、团队与管理等方面的内容。与当下工作密切相关的是这些文章的第4～6章，阐述了如何在团队设计中达成概念完整性目标。

Blaauw及Brooks（1997）的《Computer Architecture: Concepts and Evolution》这本书对IBMSystem/360（以及System/370-390-z）的架构设计和相互关系，乃至许多设计决策背后的理论都作了广泛的讨论。该书并未全面涉及设计中的过程与人工活动。但该书1.4节关于良好的计算机架构性设计标准的讨论，与这部分工作有着特别密切的关系。

设计类别

系统设计与艺术设计

这本书介绍的是关于复杂系统的设计，并且是从工程师的视角出发的。工程师关心效用和功能，但同时也注重效率和优雅。

这与艺术家和作者所做的许多设计形成了对比，他们更强调设计所带来的愉悦和所要传达的意境。当然，建筑师和工程设计师同时属于这两种阵营。

常规、适应性、原创设计

我们通常认为桥梁设计是高超的工程设计之一，在桥梁设计中，概念或者技术的突破无论从成本、功能，还是审美的角度都能带来非常明显而又具有戏剧性的影响。

然而，一大部分的高速公路桥梁都较短，推出一个50英尺的混凝土桥梁设计已是一种常规且自动化的过程。对于短桥梁，土木工程师成竹在胸，很早以前就将各种决策树、约束变量和必要条件编撰成册了。同样的情况对于为成熟的语言设计新平台下的编译器也成立。在许多领域都有这种常规的自动化设计。

这本书重点强调的是原创设计，这与因参数变更而进行的对目标的重新设计，或者对先前的设计或目标进行修改以适应新目标的适应性设计，有着明显的区别。

注释

1. Sayers（1941），《The Mind of the Maker》。
2. Brooks（1986），“No silver bullet”。

3. McDonnell（2008），《About Designing》。这本书是来自设计思想研究会议Design Thinking Research Symposium (DTRS7)的论文集。

4. Luck（2009），“Does this compromise your design?” McDonnell重印（2008），《About Designing》。

5. Blaauw和Brooks（1964），“Outline of the logical structure of System/360”。这本书描述了System/360的逻辑结构轮廓。Blaauw还把Sayer的“精神”分解成了实施和实现，我认为这一区别是非常有用的。

6. Janlert（1997），“The character of things”，这本书提出一个观点，设计是有人物性格的，并且讨论了如何做出有血有肉的设计。

7. IBM公司（1964），《IBM System/360 Principles of Operation》。

8. Vitruvius（公元前22年），《De Architectura》。

9. Pahl和Beitz（1984），《Engineering Design》。

读书笔记

- 目标
- 必要条件
- 效用函数
- 约束，尤其是预算（也许并非金钱成本）
- 决策的设计树

UNTIL（“足够好”）or（来不及了）

DO另一个设计（以提升效用函数）

UNTIL设计完成

WHILE设计方案仍然可行，

做出下一个设计决策

END WHILE

回溯设计树

找到一条之前未探索过的路径

END UNTIL

END DO

采用最优设计方案

END UNTIL

理想的设计过程模型

第2章

工程师怎样进行设计思维——理性模型

……因为设计的理论即一般的搜索理论……对象是巨大的组合空间。

——Herbert Simon(1969),《The Sciences of the Artificial》

模型概览

工程师们对于设计过程似乎有一个清晰但通常来说也是隐含的模型。这是一个关于有序过程的有序模型，也就是工程师的构思过程。我可以举一个海滨小屋设计（在第21章给出其草图）的例子来说明这是怎么回事。

目标。首先从主要目标或目的开始："某人想要建立一个海滨小屋，以享用面向大海的一块海滨场地的风浪。"

必要条件。和主要目标相关的是一组必要条件或者说是次要目的："海滨小屋应该加固，以抵御飓风来袭；它应该具备至少14个人躺卧和就座的空间；它应该为宾客提供令人难忘的视野"，等等。

效用函数。人们会根据一些效用或有用性函数来为若干必要条件依其重要性加权，以对设计进行优化。到目前为止，我知道的情况是，在大多数设计师的想象中，所有的项是由线性相加的方式组合起来的，但在单独构思每一个有用性函数时，则并非使用线性方式，而是以渐近曲线的方式趋于饱和。举个例子，必要条件之一是更大的窗户面积，这是在小屋设计中所需要考虑的问题。但是由窗户面积每平方英尺的额外增加所带来的效用是递减的。对于电源插座的数量来说，这也一样成立。窗户面积以及插座数量的总效用，看起来却仅仅是每个项的简单之和。

约束。每种设计以及每种优化都是受到一些约束限制的。其中有一些约束是二元的，只有满足或不满足的结果——"这所小屋必须位于海滨场地的边界线并再向后退至少10英尺"。其他约束则更有弹性，不过在接近限额时所付出的代价会急剧增加，比如日程表就是这样一类约束——主人可能急切地要求该海滨小屋在温暖气候来临之前完工。有些约束是简单的，比如退后尺寸的限额。另一些则在不经意间隐藏着令人生畏的复杂性——"该小屋必须满足所有的建

筑法规”。

资源分配、预算和关键预算。许多约束的形式是固定资源在各个设计要素之间的分配。最常见的是一揽子成本的预算。但是，此类约束绝不仅仅只有这么一种，而且在特定的项目中，总预算约束也并不一定就是最大限度地决定了设计师注意力的约束。例如，在海滨小屋的楼层规划中，占支配地位的定量因素是临海建筑距离的英尺数（甚至要精确到英寸）。在计算机体系结构的设计中，关键预算可能是控制寄存器或指令格式所占用的比特数，或总内存带宽的用量。而当人们解决软件的“千年虫”问题时，日程表上的工作天数成为了可分配资源中的关键项。

设计树。这么一来，按照理性模型的思路，设计师们形成设计决策。然后，在由于该决策而缩编后的设计空间中，他又形成另一决策。[1] 在每一个节点处，他都可以选取一条或多条路径，因此设计的过程可以认为是一种对于以树型结构组织的设计空间的系统化探索。

在这样一个模型中，设计在概念上（至少在概念上）是个简单的过程。人们对以树型结构组织的设计空间进行搜索，以可行性约束为依据对每种方案进行检验，从而优化效用函数。搜索算法是众所周知的，并且可以清晰地描述。

这种清晰性仅存在于对所有路径进行的穷举搜索中，其目的在于寻找一个真正的最优解。设计师们通常只去寻找一个“足够好”的最低限度满足解。[2] 许多工程师似乎采用了某种深度优先策略进行近似估算，并在每个节点上选择最有前途或最有吸引力的方案，并采用探索到底的办法来达成目的。如果遇到死胡同，他们会采用回溯的办法并尝试另一条路径。预感、经验、连贯性和审美旨趣引导着每一次的方案选择。[3]

该模型的构思从何而来

将设计过程建模为一种系统化的、按部就班的过程的观念，似乎肇端于德国机械工程社团。Pahl和Beitz在他们7次修改其稿的伟大论著中阐发了目前被最广泛地接受的观点。[4] 他们对达·芬奇（1452—1519）的《Notebooks》中关于设计备选方案的系统化搜索过程施以实践观点的考察，而并非只泛泛阅读那显式写出的陈述。

Herbert Simon在其著作《The Sciences of the Artificial》（1969, 1981, 1996）中独立地提出设计就是一个搜索过程的主张。他提出的模型及相关讨论远比这里的要复杂。Simon乐观地认为设计过程就是搜索人工智能意义下的合适标的（只要有足够的处理能力到位），他也投身于严格化理性设计模型的筹划，因为这样一种模型对于设计过程自动化而言乃是不可或缺的先驱力量。他的模型仍然有影响力——即使到了今天，我们已经认识到，其原始设计中的“险恶问题”[5] 可以说是在人工智能中最没前途的候选之一。

在软件工程领域，Winston Royce对于因为采用“先写了再说”的方法而造成的大型软件系统的项目失败而深感震惊，于是独立地引介了一种由7个步骤组成的瀑布模型，以将流程加以整顿，如第3章的第1插图所示。事实的情况是，Royce是把他的瀑布模型当做一个假想的批

评对象提出来的，但是有很多人已经引用并追随这个假想的批评对象，他提出的更为复杂精妙的模型反而被晾在一边儿了。我在年轻的时候也犯过那样的错误，并在之后公开地为其忏悔。[6] 即使有那么一点儿讽刺的味道，Royce的7步模型仍然必须看做是设计的理性模型的基础性表述之一。

Royce强调，他的7个步骤是彼此泾渭分明的，需要分别地规划并各有专人负责。其中确有重叠的部分，但这部分被仔细地限定在一定范围之内：

各个步骤的顺序安排乃是基于以下的概念：每前进一步，设计就变得更加详尽，在（邻接的）前一步和后一步之间有一定的重叠，但是在序列中距离较远的步骤就不太会有什么重叠之处了……我们拥有一种有用的退路，这往往可以将早期工作中仍然可资利用的以及得到保留的部分尽可能地最大化。[7]

设计空间可以表达为树型结构的观念，是在Simon的著作中隐含地提出的。这个观念在Gerry Blaauw和我合著的《Computer Architecture》一书中有具体的描述和图解。[8] 在该书中，我们将处理器架构的设计方案以严格的层阶架构形式组织在一个巨大的树型结构中，以83个链接子树来表示。有关闹钟的设计树可以作为一个简单的例子，如图2-1所示。其中，人们可以看到两种分别以开放和封闭的根型表示的分支。第一种，如“闹铃”节点所示，表示的是细分单元，每一个分支都是一种特定的设计属性，且必须指定其值。此即所谓属性分支。而备选方案分支，由“铃声”节点所示，则枚举了所有的备选方案，人们必须从中选择适当的方案。

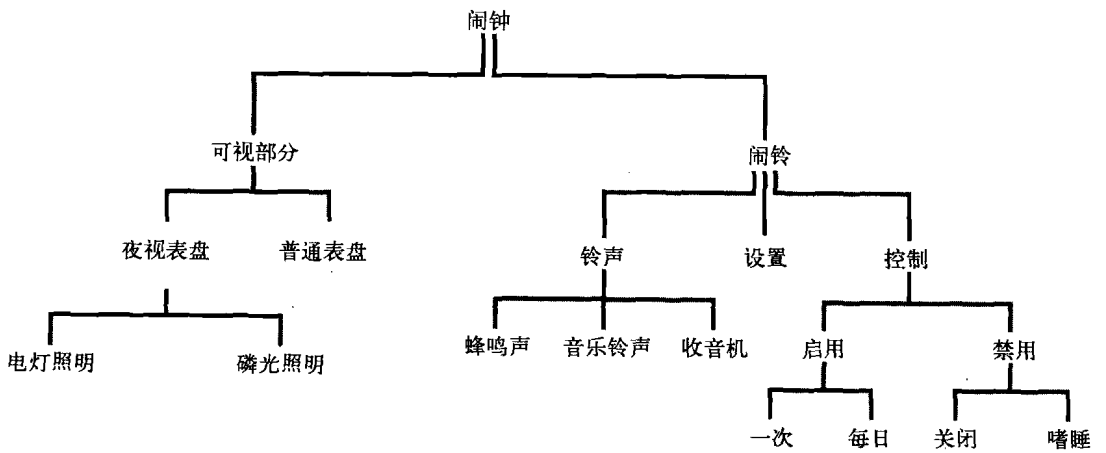


图2-1　闹钟的设计树（部分），选自Blaauw和Brooks(1997)，所著的《Computer Architecture》的图1-12和图1-14

理性模型有哪些长处

与“先开始编码再说，先开始构建再说”的行为相比，任何将设计过程系统化的工作都可以视为一种长足的进步。它为设计项目的规划提供了清晰的步骤。它为日程规划和进度评估定义了明确的阶段里程碑。它为项目组织和人员配备指明了方向。它改进了设计团队的内部沟通。

而在设计团队和其项目经理之间以及项目经理和其他利益攸关者之间而言，它对于沟通的改进尤为显著。新手很容易就可以上手。掌握了它，新手在面对分派给他的第一个设计任务时，就知道从何入手了。

理性模型在特定的情形下会体现出更多的长处。在项目早期就给出目标的显式陈述、相关的必要条件以及约束说明，这有助于避免让团队陷于举棋不定的局面，也促使团队形成关于项目宗旨的统一认识。在开始编码或正式的制图工作开始之前做好整体的设计过程规划，就能够规避大量麻烦，也避免让许多努力付之东流。将设计过程打造成对于设计空间的系统化搜索，可以拓宽设计师个人的眼界，并把他们的视界提升到远远超过其先前的个人经验的程度。

不过，理性模型太过简化了，即使是Simon洋洋洒洒、高度成熟的版本也不免于此。因此，我们必须对其缺陷加以审查。

注释

1. 按照Simon(1981)《The Sciences of the Artificial》的习惯，在整本书中我采用“man”作为一个一般性的名词加以使用，两种性别都包括在其指代的对象中，同样“he”（他）、“him”（他的——形容词用法）和“his”（他的——名词用法）也一律作为兼具两性的代词。我觉得继续使用符合传统的，把女性和男性平等地置于这些一般性的代词指代之中的做法十分亲切，这好过生硬地使用一些画蛇添足的，因而也是分散人们注意力的噱头。

2. 寻找“最低限度满足解”就是找到足够好的解，而并不一定是优化解（Simon(1969)，《The Sciences of the Artificial》）。

3. 但是参见Akin(2008)的“设计中的变量与不变量”，它从DTRS7协议中发现证据表明建筑架构设计师们往往会在各个层次中横向地搜索若干个备选方案，而工程设计师们则主张从初始解决方案的提案出发，展开深度优先搜索。

4. Pahl和Beitz(1984)，《Engineering Design》。

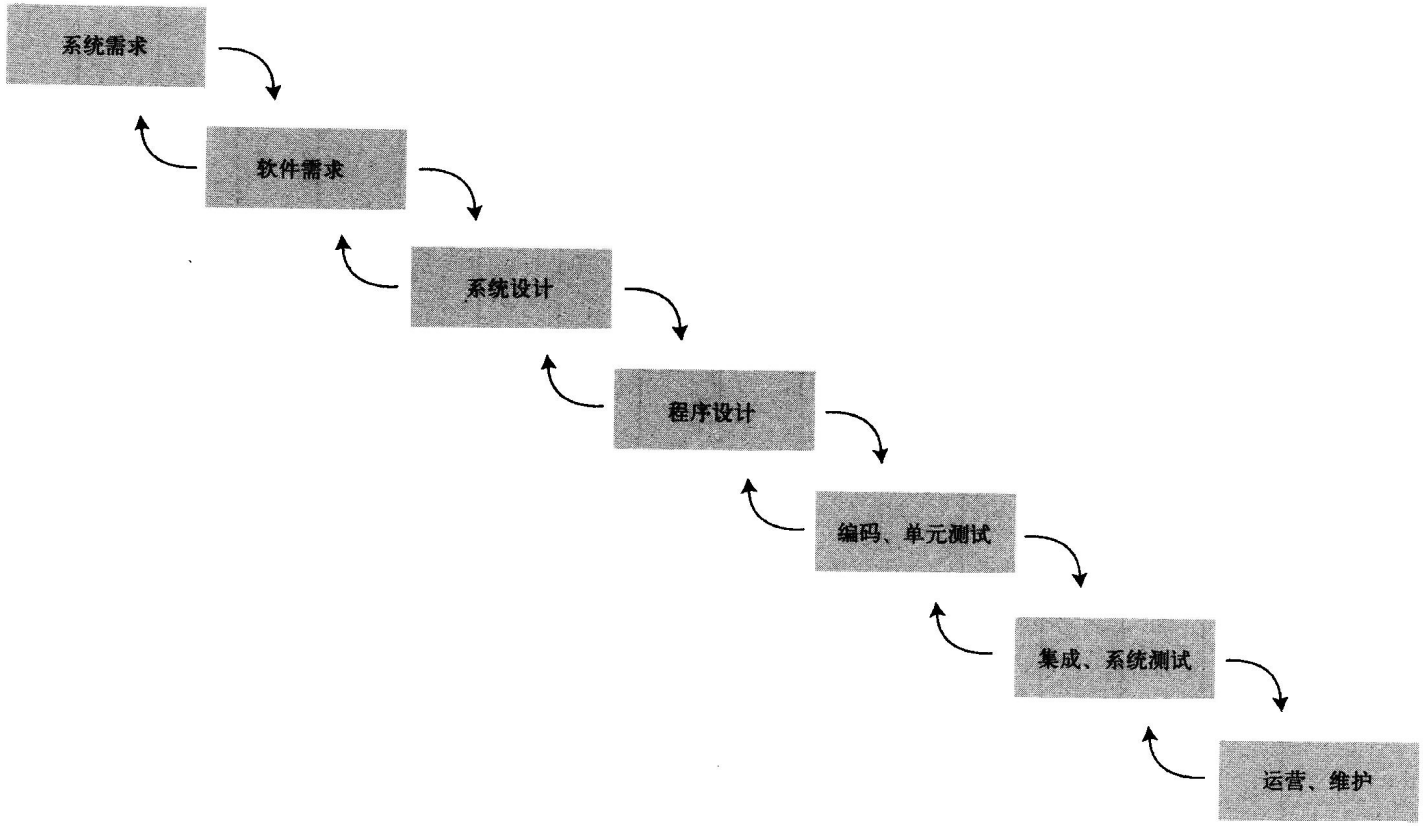
5. Rittel和Webber(1973)的“规划的一般理论中的困境”，它正式地定义了这个词。它也在以下的词条中有着详尽的讨论：http://en.wikipedia.org/wiki/Wicked_problem。

6. Brooks(1995)，《人月神话》，265。

7. Royce(1970)，《大型软件系统开发的管理》，329。

8. Blaauw和Brooks(1997)，《Computer Architecture》。

读书笔记



软件开发的瀑布模型

参照Royce(1970)，“Managing the development of large software systems”和Boehm(1988)，“A spiral model of software development and enhancement”

第3章

理性模型有哪些缺陷

现实情况是，设计师只把理性模型视为一种理想化的东西。它以某种方式描述了在我们的认识中设计过程应该如何运作，但在现实生活中，并不是那么一回事。

事实上，不是每个工程师都会大方地承认在他的心目中有这么一个很天真很理想的模型。但我认为我们中的大多数人都有这样的想法，我自己心中的这种想法持续了很长时间。因此，让我们对理性模型进行仔细彻底的剖析，以确切地了解它究竟在哪些方面脱离了现实。

我们在初始阶段并不真正地知道目标是什么

理性模型最严重的缺陷在于，设计师们往往只有一个模糊不清的、不完整的既定目标，或者说是主要目的。在此情形之下：

设计中最困难的部分在于决定要设计什么。

在我还是学生的时候，有一个暑假里去替一家很大的军火商打工，在那里我被指定去做设计和构建一个小型数据库系统的工作，用以跟踪某个雷达子系统的上万张图纸以及其中每一张图纸的更新状态。

过了几个星期，我做出来了一个能运行起来的版本。我自豪地向我的客户演示了一个输出报告的样例。

“做得不错，这的确是我想要的，不过你可否把这里改一下？那样我们就可以……”

在接下来的数个星期，每天早上我都给客户演示输出报告，每次都是顺应了前一天提出的要求之后的修订结果。每天早上，他都会对产品报告研习一番，然后使用一成不变的、彬彬有礼的口头禅提出另一项系统修订的要求。

系统本身很简单（是在打孔卡片机上实现的），而且那些修订在概念上看起来也是平淡无奇的。就算是最影响全局的变化也只是将图纸列表按照内部等级排序或缩进，而等级是用卡片上单独一个0～9的数字来表示的。其他的改进包括多级局部汇总——当然有例外情况要处理——以及自动地为不同的值得注意的值标注上星号。

有那么一阵子，我很是愤愤不平："为什么他不可以就想要的内容下定决心？为什么他不能把想要的对我一口气说完，而偏偏要每天挤一点出来呢？"

然后，我一点点地认识到，我为客户提供的最有用的服务，乃是帮助他决定什么是他真正想要的。

那么，如今的软件工程原则要复杂得多了。我们认识到，快速原型是一种进行精准的需求配置的必要工具。不仅整个设计过程是迭代的，就连设计目标的设定过程本身也是迭代的。

软件工程领域的复杂化不仅没有停止的势头，甚至连明显的放缓也看不到，在汗牛充栋的文献资料中，"产品需求"仿佛是给定设计过程的常规假设前提。不过，我要提出一点异议，那就是，在初期就能了解整个产品需求属于相当罕见的例外，而远非常态：

设计师的主要任务乃是帮助客户发现他们想要的设计。

至少在软件工程领域，快速原型的概念有其地位及其公认的价值，但在计算机（体系结构）设计或是建筑架构设计中，它的地位与在软件工程中并不总是相当。但无论如何，在目标迭代的方面，我在这些设计领域都看到了相同的现象。越来越多的设计师们为计算机构建模拟器，为建筑构建虚拟环境演练，作为快速原型，以促成目标的收敛。目标的迭代必须作为设计过程的固有组成部分加以考虑。

我们通常不知晓设计树的样子——一边设计一边探索

在复杂结构的初始设计，如计算机、操作系统、航天飞机以及建筑等，每一项主要设计工作在如下方面都有足够的新意：

- 目标
- 必要条件和效用函数
- 约束
- 可用的加工技术

这些步骤中，设计师很少有机会能坐下来先验地绘制出一个设计树来。

此外，在高技术领域的设计中，甚至很少有设计师能够拥有足够的知识以绘制出该领域中基本的决策树来。设计项目往往会进行两年以上。设计师在此期间会得到升迁，从而脱离一线的设计工作。这样导致的后果就是，很少有设计师会在其职业生涯中将其一线工作深入到参与上百个项目的程度。这意味着，对于设计个人来说他就失去了探索其设计科目的所有分支的宝贵机会。因为这是工程领域的设计师的特点，和科学家大相径庭的是，他们很少会去触碰那些不能一眼看出是通往解决方案的备选途径。[1]

相反地，设计师们会一边做着设计，一边进行设计树的探索——做出某个决策，然后查看由它启发或否决的备选方案，继而依此做出排在下一个的设计决策。

（设计树上的）节点实际上不是设计决策，而是设计暂定方案

事实上，特定的设计树自身只是在树形结构中搜索的简化模型。如图2-1所示，有并列的属性分支，也有备选分支。在一个分支中的各个备选方案彼此紧密联系——或彼此相斥或相辅相成或平分秋色。我们在《Computer Architecture》一书中给出的大块头设计树其实还是过分简化了；那样的一个设计树中所展示出来的“计算机众生相”对于阐明决策之间的联系乃是必不可少的。[2]

这意味着，在设计树的每一个节点处，设计师所要面对的不仅仅是为单独一个设计决策准备的若干简单备选方案，而是为多个设计暂定方案准备的备选方案了。

此外，设计树中的决策排列顺序事关重大，可以参见Parnas在其经典论文“Designing software for ease of extension and contraction”中所阐述的真知灼见。[3]

以树型结构表示的设计模型，其复杂性带来的组合爆炸是人们思维中的难以承受之重。（这情形就像是国际象棋中的棋子移动所构造出来的状态空间树。）该困境在第16章会有进一步的探讨。

有用性函数无法以增量方式求值

理性模型的假定是，设计是对于设计树的搜索，并且在每个节点处人们可以对若干下一级分支的有用性函数求值。

事实上，除非探索到所有分支的所有叶节点的程度，否则人们就很难做到这一点，因为大量的有用性指标（比如性能、成本等）对于随后的设计细节有着强烈的依赖。因此，虽然对有用性函数的求值在原则上是可行的，但是在实践上，人们就会在这里再次遭遇组合爆炸。

那么，设计师该怎么做？估算！理所当然，正式的也好，非正式的也罢，都要做估算。在求精的步骤中，人们必须对设计树进行剪枝。

经验。很多辅助信息都能够促进该过程中的直觉判断。其中之一是经验，无论是直接还是间接的：“OS/360的设计师们将OS/360操作系统中在整个系统范围内共享的控制块的格式细节暴露出来，这后来成为了维护工程师的噩梦。我们会将其封装为对象。”“宝来B5000系列在很久以前就探讨过基于叙词的计算机体系结构。由于本质性能损失实在太大，我们不打算继续深入设计子树了。”当然，工艺方面的权衡早已日新月异，但是上面的例子仍然很好地说明了经验教训。研习设计史的最有力的原因是去了解怎么样的设计方案是行不通的以及为什么这些设计方案行不通。

简单估算量。设计师们经常在进行设计树探索的早期就例行地采用简单估算量。建筑师们在得知目标预算以后，会粗略地估算一个平均到每平方英尺的成本，得出一个每平方英尺的目标，并使用它进行设计树的剪枝。计算机架构设计师们则会根据指令组合来对计算机性能做粗略的初步估算。

当然，这样做的危险在于，粗略的估算有可能会将本来可行但由于某个特定的估算量所采用的估算方法而看上去不可行的分支剪掉。我见过一个建筑师，他以过高的成本为理由，把一个早已指定的房顶结构之下的一堵墙壁给取消了，纯粹基于例行的平方英尺估算量他就作了这样的决定。而实际情况是，为增加的空间而付出的成本主要在于房顶，但那个是已经计算在内的了，所以这么一来边际成本会非常低。

将欲免费取之，必先无偿予之。

必要条件及其权重在持续变化

Donald Schön，已故麻省理工学院的都市研究与教育教授、设计理论家如是说：

（当设计师）按初始状况进行设计改造的时候，状况本身会“抵触”，而他只能就这种状况反弹做出回应。

在健康的设计过程中，这种状况交互是自反的。在回应状况反弹时，设计师会将问题的构造、行动的策略以及现象的模型纳入行动的考量，在每一步的推进中都隐含了这些考量。[4]

简而言之，在对于权衡的沉思中，一种对于整体设计问题的新理解逐渐浮现，即它是诸多因素以错综复杂、彼此牵制而又彼此交互的方式组合的结果。由此，对于诸项必要条件的权重计算方法就发生了变化。客户方——如果有——也逐渐地接受了这种理解，以此为出发点来形成对他将得到的成果的期望以及他将如何使用这个成果的预见。

例如，在我们的房屋改造设计中（详见第22章），一个在原始项目中看似简单的问题，在设计推进的过程中突显出来，原因就在于我和我的妻子将用例场景应用到了原始设计时的一个发问：“来参加会议的客人们该将他们脱下的外套搁在什么地方呢？”这个看起来权重不高的必要条件产生的影响规模很大，结果是把主卧从房间的一端迁移到了另一端。

此外，对于那些必须进行分块加工的设计，比如建筑和计算机的设计，设计师们从建造者处一点一滴地学习到有关设计和加工是如何交互的理解。大量的必要条件和约束条件被变更和改进。加工工艺也会有演进的过程，这对于计算机设计而言就是老生常谈的事了。

由于许多必要条件（如速度）是以性价比为权重的，这就会导致另一种现象的发生。随着设计向前推进，人们会发现在只需负担极少的边际成本的前提下，就可以增加某些特定的有用性的机会。在此情形下，在原始的必要条件清单中根本不存在的项目就会被添加进来，而这往往会使其后的设计变更中要求保留的预算余地被挤占。

例如，只有北卡罗来纳大学的西特森厅在设计、建造和投入使用的过程中，计算机科学系作为该建筑的用户，才学会如何在由楼下大堂、楼上大堂、学院会议室、讲演厅和走廊的成套空间内，将所有这些漂亮地组合成一个能够举办多至125人参加的会议的基础设施，同时把因其施工而对大楼内其他工作的影响减至最低。这个成功也可谓有着各种机缘巧合，因为在最初的建筑方案中并未考虑该厅所拥有的功能。然而，这是价值颇高的特色：任何对于西特森厅的

未来修订肯定会把保留这些功能作为目标。

约束在持续变化

即使设计目标固定而且已知，所有的必要条件皆已枚举清楚，设计树已经刻画成很精确的样子，并且有用性函数也有着明确无误的定义，设计过程也仍然会是迭代的，因为约束在持续变化。

通常情况下是环境变了——市政厅会通过令人沮丧的规定给设计投下新的阴霾；电气规范每年都会更新，本来计划要用的芯片被供应商召回了，等等。一切都在不断变化，即使在我们的设计向前推进的过程中也并未留步。

约束也会由于设计过程中甚至加工过程中的新发现而发生变化——建筑工人碰到了无法凿穿的岩层，分析结果表明芯片的冷却问题成为了新近的约束，等等。

并非所有的约束变化都是增长型的。约束也经常消弭于无形。如果这种约束变化是偶发的，而不是人为的，熟练的设计师就能利用这样的新机遇，发挥其设计的灵活性，以绕过该约束。

并非所有的设计都有灵活性。更为常见的是，当我们深入一个设计过程时，就意识不到原来某个约束已经消失不见，也想不起来由于它的消失而早已排除的设计备选方案了。

重要的是要在设计过程的一开始就明确地列出已知的约束，作为架构师所谓的设计任务书的组成部分。设计任务书是一个文档，需要与客户共同完成，它规定了目标、必要条件以及约束。本书的网站给出了一个设计任务书的示例。设计任务书和正式需求描述文档不是一回事，后者通常是具有合同约束力的、定义某个设计方案的接受标准的文档。

将约束明确列出，是把丑话说在前面，这就可以避免日后突然爆发令人不快的局面。这同时也是在设计师的脑海中烙下对于这些约束的印象，从根本上提高某一约束消失时被设计师发现的可能性。

我们都是围绕着约束来做设计的，该过程要求对于设计空间中少有人问津的犄角旮旯有着很多创新和探索的精神。这是设计之乐的部分所在，这也是设计之难的大部分所在。

在设计空间之外的约束变化。然而，有时，设计的突破性进展来自于完全跳出设计空间的囚笼，在那里将设计的约束加以消除。在设计厢房的时候（见第22章），我努力了很久均未果，就为了一个令人心情灰暗的靠后尺寸需求约束以及音乐室的必要条件（要放置两架三角钢琴、一架管风琴以及一个正方形的空间以容纳弦乐八重奏乐队，加上一英尺宽的教学之用的余地）。如图3-1所示，这是设计过程的一次迭代以及其约束。

这个设计过程中遇到的棘手问题最终是在设计空间之外得到了彻底解决——我从邻居处买下了另外五英尺的地皮。这比向市政厅申请靠后尺寸变更——另一种设计空间之外的解决途径——来得可能更经济，并且肯定更富效率。它同样给设计方案的其他部分带来了解放，对于F书房的西北角的定位贡献尤其明显（见图3-2）。

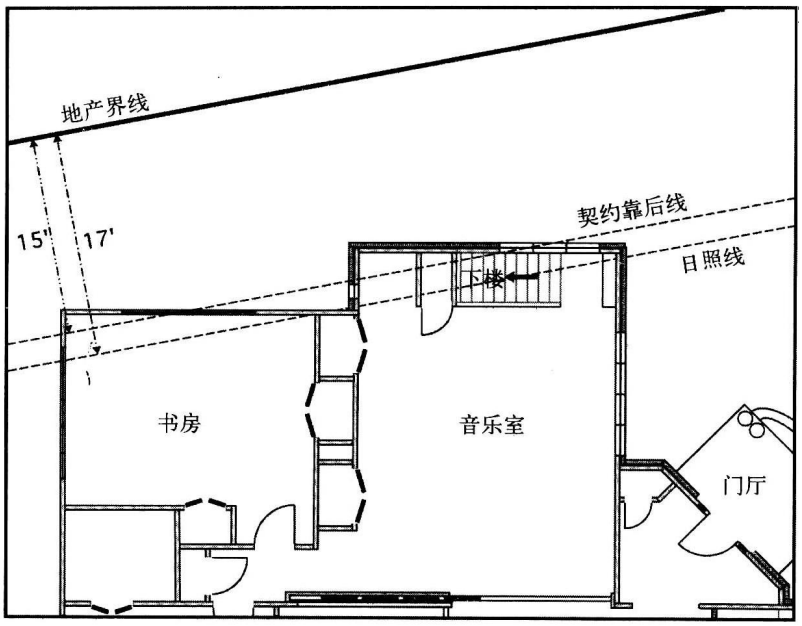


图3-1　依约束进行的设计

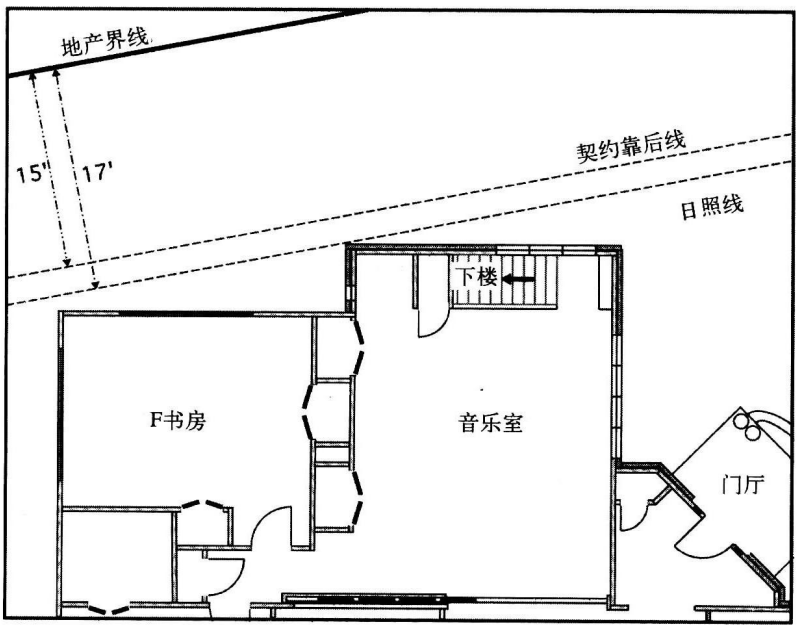


图3-2　约束被放松了

将设计任务书中的已知约束明确列出的好处，在此处也有体现。设计师们可以定期地检视这个清单，自问："现在有些东西已经变化了，这个约束能够去掉了吗？能不能通过在设计空间之外想出办法来规避它呢？"

对于理性模型的其他批评

理性模型是一种自然的思维模型。理性模型，如上所述、如上所评，似乎看上去相当幼稚。但它是人们能够自然而然地想到的一种思维模型。其思维自然程度可以从Simon版本、瀑布模型版本以及Pahl和Beitz版本分别独立地提出而得到强烈的印证。然而，从最早的时候开始，设计界就有了对于理性模型有说服力的批评。[5, 6, 7]

设计师们根本不那样做事。也许对理性模型最具解构性的批评——尽管也许亦是最难以证明的——就是经验最丰富的设计师根本不那样做事。虽然已经发表出来的批评只是偶尔才会有"皇帝没有穿衣服耶！"这样指出该模型并未反映出专业实践做法的呛声，但是人们还是可以感觉到不厌精细的分析背后的这种压倒性的主张。[8]

Nigel Cross，其绅士般的言论，也许是最具张力的不同意见。引经据典之下，他毫不讳言地说：

*有关问题求解的习惯思维，往往看起来和专家级设计师们的行为背道而驰。不过，设计活动和使用习惯思维进行问题求解的活动有着相当多的不同之处……我们必须在从其他领域引入设计行为模型时倍加小心。对于设计活动的实证研究经常会发现，"有直感力的"设计能力乃是最有成果的，也是和设计的内廓性质最密切相关的。不过，**设计理论的有些方面就是企图针对设计行为开发出反直觉的模型和处方来。**[9]*

又及，

在绝大多数设计过程的模型中都忽视了性质上处于同等地位的设计推理。在有着公论的关于设计过程的模型中，比如说由德意志工程师协会颁布的模型中（VDI，1987）……就主张设计活动应该分成一系列的阶段进行……在实践中，设计活动似乎是以在子解域和子问题域的区间振荡的方式进行，同时也是将问题分解，尔后合并所有的子解决方案的过程。[10]

我发现，这个争鸣意见本身以及其实证材料都很具说服力。此处提及的振荡的确可以说是我所有设计经验的特点所在。"外套在哪里摆放？"这样的需求发掘了我们房屋设计过程的深层次内容就是个典型例证。

Royce对于瀑布模型的批评。Royce在他的论文原稿中描述了瀑布模型，以便他能够演示其不足之处。[11]他的基本观点在于，尽管在毗邻方块之间有反向箭头表示逆向的工作流，但是该模型仍然行不通。不过，他的对策仅仅是采取了容许逆工作流箭头指回两个前向方块的模型罢了。治标不治本。

Schön归纳的批评小结。

[Simon]发现在专业知识和现实世界的要求之间有着一道鸿沟……Simon提出……采用科学化设计的方式弥补这个鸿沟，他的科学化方法只能应用于对从实践中总结的、良好构建的问题。

如果这种所谓的技术化理性模型未能考虑到实践能力的“发散”情景，用了还不如不用。那么，还是让我们转而追寻一种基于实践的认识论，它隐含在艺术的、直观的实践过程中，而这样的过程已经被一些实践者用以应对不确定性、不稳定性、单一性和价值观冲突。[12]

但是，尽管有这些缺陷和批评，理性模型仍然不屈不挠地存在

通常来讲，某种理论或技术的原始创意提出人都比后继的追随者更了解其承诺所在、其局限所依以及其正确的应用范围。由于天资有限、热忱有余，他们的一腔热情却导致了思维僵化、应用偏差和过分简化等。

遗憾的是，理性模型的诸多应用亦是如此。近至2006年的文献，设计研究员Kees Dorst亦不得不承认，

尽管从彼时到现在已经有了长足的进步，但是Simon所著的探讨问题求解以及具有病态结构的问题本质的原始著作，仍然在设计方法论领域有着难以忽视的影响。基于Simon提出的概念框架的理性问题求解范式，在该领域仍然占主导地位。[13]

诚哉斯言！在软件工程领域，我们仍然太过盲从瀑布模型——理性模型在我们领域的衍生品。

德意志工程师协会标准VDI-2221。德意志工程师协会于1986年采纳了理性模型，本质上如同Pahl和Beitz所介绍的那样，作为德国机械工程业界的官方标准。[14]我见过很多由于这场运动引发的思想僵化。但Pahl自己一直在尽力设法澄清如下：

在VDI准则2221-2223以及Pahl和Beitz（2004）中给出的过程并非“按部就班”式的，它只能被视为有明确目的的基础行动的指导性准则。在实际行动中有用的解决方案可能或是选择一种迭代途径（即那种带有“前进或回退”步骤的途径），或在采用更高信息层次上的反复途径。15

美国国防部标准2167A。类似地，美国国防部于1985年将瀑布模型正式纳入其标准2167A。[16]直至1994年，在Barry Boehm的领导下，他们才开放了其他模型的准入门槛。

那又如何？我们的设计过程模型真的那么事关紧要吗

为什么就过程模型讲了这么多？我们或是别人用来进行设计过程的思维真的会影响我们的设计本身吗？我认为的确是这样的。

并非所有的设计思想家都同意我的观点。剑桥大学教授Ken Wallace与Pahl和Beitz著作的所有三个版本的英文译者，相信一个主要的进步会是某种让人能够轻而易举地理解和沟通的模型。他指出这一点对于设计的初学者来说是多么重要。Pahl和Beitz的模型为新手做设计准备了一个入手的空间，这样他们就不会徒陷彷徨。“我把Pahl和Beitz的图（他们书中的图1-6）放在

我的讲义上，并解释了一下。在我紧接着下一张幻灯片上就写道：‘但现实中设计师是不那样工作的。’”[17]

不过，我担心是否有更年轻的、个人设计经验更少的教育工作者总是会这样说。

苏珊娜·罗伯逊和詹姆斯·罗伯逊夫妇有着国际化咨询的实战经验，并且著有需求规划的重要作品，他们认为理性模型的不足之处并不值得大惊小怪。“更了解设计的人也更有智慧。”[18]

无论如何，我相信我们带有缺陷的模型以及对其的盲从，将导致臃肿、笨重、功能过多的产品以及时间表、预算和性能的灾难。

右脑型的设计师。绝大多数设计师是右脑型的，是视觉-空间导向的。事实上，我在考察未来设计师的天赋时，用于投石问路的问题就是：“下一个11月在什么地方？”当听我说话的人显出莫名其妙的表情时，我会进一步地解释，“你有一个日历的空间思维模型吗？很多人都有的。如果你也有，能给我描述一下吗？”几乎每一个有竞争力的候选人都会有这样一个模型，但这些模型彼此大相径庭。

类似地，软件设计团队总是在他们共用的白板上画草图，而不是写文字和代码。而建筑师则把在描图纸上用的美工笔视为一个不可或缺的沟通工具，但是在独立思考时则用得更多。

因为我们设计师是空间思维导向的人，我们的过程模型在脑海中是以图表的形式深深植根的，无论其具体形式是Pahl和Beitz的垂直式矩形，还是Simon的树型结构，抑或是Royce绘制和批评的瀑布状图形。这些图表在潜意识层面大大地影响着我们的思维。因此，我认为一种先天不足的过程模型会以我们不能完全知晓、只有一知半解的方式阻碍我们前进。

一个由于采用了理性模型而造成的明显损害就是我们无法对接班人进行恰当的教育。我们教给他们连我们自己都不遵循的工作模式。结果，在他们形成自己在现实世界中采用的工作模式的过程中，我们就没能提供有用的帮助。

我认为，对于更资深的，尤其是那些有在业界设计经验的教师来说，情况就会大不一样。我们很清醒地认识到，这些模型是有意简化过，以帮助我们解决真实生活中遇到的问题，后者往往复杂得令人生畏。因此，我们在教给学生的时候也会提出“这只是地图，而不是真实的地形”的警告，因为只有模型是不够的，即使在可以适用的场合，它们也仍然有失精确。

在软件工程实践领域，还可以很容易地发现另一种不利因素——理性模型，无论以何种面目出现，都会导向一种先验的设计需求描述。它导致我们盲目相信这种需求真的是可以预先制订的。它也导致我们在对于项目一无所知的基础之上就彼此签订了合同。一种更加现实的过程模型将使得设计工作更富效率，并省却许多客户纠纷和返工努力。第4章和第5章将阐述需求问题。

瀑布模型是错误的、有害的，我们必须发展并摒弃之。

注释

1. 工程师需要的是最低限度满足解，而科学家则需要的是发现，这往往可以通过在更大范围里探索而求得。

2. Blaauw和Brooks(1997)，《Computer Architecture》，26-27，79-80。

3. Parnas(1979)，“为简化可伸缩性软件而进行的软件设计”，明确地将设计过程作为树型结构的遍历来处理。他强烈主张使设计尽可能地灵活。他敦促人们设计的灵活性是重要的目标之一。在软件工程领域，面向对象的设计也好，敏捷开发方法也好，都将此作为根本目标之一。

4. Schön(1983)，《The Reflective Practitioner》，79。

5. 出乎意料的是，我发现对于Pahl和Beitz进行理性模型的构造方法，以及对于Simon对此的大部分构造方法都少有批评。Pahl和Beitz自己倒是意识到了该模型的不足之处：在他们著作的后续版本里，他们的模型（在第2版、第3版英文版中的图3-3、图4-3）包括了越来越多的迭代步骤（Pahl和Beitz(1984, 1996, 2007)，《Engineering Design》）。Simon三个版本的《The Sciences of the Artificial》并未反映出计划中模型的变化，尽管他于2000年11月在和我的私人谈话中曾经透露，他自己对该模型的认识已经有所改变，但是他还没有机会去就此重新思考和改写原著。

Visser(2006)，《The Cognitive Artifacts of Designing》，该书中的9.2节是精彩的一节，“Simon在其更新近工作中的微妙位移”，它考察了Simon在其更近期发表的论文中体现出来的思想演化。

Visser与我一样吃惊地发现，这些思想演化并未反映到《The Sciences of the Artificial》的更新版本中。

6. Holt(1985)，“设计还是问题求解”：

有关工程设计存在两种截然不同的解释。一种是“问题求解”解释，这在很多大专院校中比较普及，它强调使用标准化技术对结构化的、有明确定义的问题求解，这种解释可以追溯到“裸”系统思维。另一种是“创新设计”解释，它将分析思维及系统思维和人为因素结合在工程设计中，以创设和利用各种机会来更好地为社会服务。本论文旨在探讨“问题求解”解释在应付很多现实世界的任务时会遇到的种种限制。

7. 如果说Cross的批评是基于实证的，那么Schön的批评则针对理性模型的深层哲学。他说，理性模型，正如Simon所阐明的那样，是一种更加普适的他称为技术理性的哲学思维方式的自然外延，他认为后者继承了现在已经不再有市场的实证哲学的衣钵。他发现只立足于这种深层哲学本身对于理解设计的要求而言是完全不够的，尽管它已经被制度化地引入了最专业化的设计课程中：

从技术理性的角度来看，专业设计是个问题求解的过程。问题……通过从可用的手段中找到最符合既定目标的选择而得以解决。但是由于过分强调了问题求解这回事，我们忽视了问题本身的要求，以及定义所作决定、最终达到的目标，以及选择可能的途由等这些过程。在现实世界的实践中，问题在实践者那里往往并不像乍看之下那样表现。问题必须由带来问题的情形

材料构造出来，而后者往往令人费解、麻烦不断而且难以捉摸……实践者必须得就此费些力气才行。他必须得把一开始完全不显山露水的情形搞个水落石出……像这样的情形才被专业人员越来越多地视作他们工作的核心……技术理性取决于如何看待目标这回事。

8. 一个生动的例子是Seymour Cray在1995年所言节录："我应该算是个科学家，不过我在作决定时使用直觉更胜过考虑逻辑。" http://www.cwhonors.org/archives/histories/Cray.pdf，访问于2009年9月14日。

9. Cross(2006)，《Designerly Ways of Knowing》，27

10. Cross(2006)，《Designerly Ways of Knowing》，57。Dorst(1995)，"描述设计活动的图表比较"，有一个特别好的关于Simon和Schön的比较讨论。他们的杂志文章在Cross(2006)，《Analysing Design Activity》中重印。Dorst还展示了在代尔夫特Ⅱ协议中，Schön的模型和观察所得的设计师行为更精确地符合。

11. Royce(1970)，"大型软件系统开发的管理。"

12. Schön(1983)，《The Reflective Practitioner》，45-49。

13. Dorst(2006)，"设计问题和设计悖论。"

14. VDI(1986)，《VDI-2221：Systematic Approach to the Design of Technical Systems and Products》。

15. Pahl(2005)，"VADEMECUM-设计方法论之开发和应用中的建议。"

16. DoD-STD-2167A企图修正此问题，但遗憾的是它将瀑布图表放到了一个突出位置，从而一切都还是基本保持不变。MIL-STD-498取代了2167A，并着手处理了模型问题。DoD自采纳了工业标准IEEE/EIA 12207.0、IEEE/EIA 12207.1和IEEE/EIA 12207.2之后才取代了498。

17. 个人通信（2008）。

18. 个人通信（2008）。

波音-西科斯基RAH-66卡曼契直升飞机（原LHX）西科斯基飞机制造公司/Richard Zellner/美联社图片

第4章

需求、罪念以及合同

任何在项目伊始就规划所有的可能需求之企图都会落败，并以可观的延误告终。

——Pahl和Beitz(2007)，《Engineering Design》

委员会认为，若要达成有着清晰、完整状态的系统级需求，就要求在首个和第二个阶段里程碑之间与潜在承包商之间进行交互。

——James Garcia，《Pre-Milestone A and Early-Phase Systems Engineering》

呈阅美国空军研究委员会部分

一段恐怖往事

某将军曾经在海军陆战队当过航空兵，对直升飞机很了解。他和我一起被分派到五角大楼内部并成立了一个国防科学委员会的下属委员会。我们专心听取了一名上校对于进行中的实验性轻型直升飞机（LHX，卡曼契军用直升飞机）设计的简要介绍，该飞机是下一代轻型攻击型直升飞机，将耗用数十亿美元的经费，且大兵们的性命概系于此。该直升飞机将成为现有的四种不同的直升飞机的继任者，而这四种飞机是用来执行不同的任务的。

上校概述了由一个跨组人员组成的委员会制订的需求，它反映了好几个用户组的意见：

“须能以某时速飞行某距离。须能携带X装甲，装备Y武器，携带Z数量的弹药，携带W名全副武装的战士，乘杂人等另计。”

“须能近地飞行，低于雷达覆盖面。即使在漆黑的暴风雨之夜，亦能爬升以避开障碍物。须能实施跃起射击，并趴降以避开回击火力。”

然后，没有任何语气和表情变化，他又说，“并且，它须能自行横渡大西洋（这个距离超过了其常规作业范围）。”

将军和我都露出了十分震惊的神情。上校见状马上回应道：“哦，它的设计将考虑到这一点，办法是取下所有的军械和弹药，并将多出来的空间塞满成桶的汽油。”但是，我们的怀疑在于其设计原理，而非其可行与否。即使是我——一个计算机设计师，从直觉上也知道人们必

须为这种能力付出代价，不是以金钱的形式，而是必须以削弱其他部分的能力的方式来补偿。

“为什么它非要具备渡洋的能力不可？显而易见，它只需要渡洋两次，如果你走运的话。”

“我们没有足够的C-5运输机来把他们运送过去。”

“那我们为什么不挪用一部分LHX的项目经费来购买更多的C-5运输机，而非要在LHX的设计上采取折衷呢？”

“因为那办不到。”

我们对于这种需求的极端性的震惊完全不亚于对于我们的上校在这种极端性之下显得满不在乎的态度的震惊。也许我们的直觉是错误的。也许，自运送能力的边际成本确实很低。也许掌握了相关知识的设计师已经在这个问题上进行过一番苦战。

但我们随后的谈话并不令人兴奋。如果我没记错，LHX的需求委员会中既未包括航空器设计师，也未包括直升飞机的飞行员——仅仅是由一堆善于拿他们各自组内的东西出来在跨组谈判中卖弄的官僚们组成罢了。[1]

殊为不幸，无独有偶

许多读者都会毫无困难地在脑海里浮现出LHX需求委员会的开会画面：我们都出席过这样的会议。

每个列席人都会给出一个愿望列表，该列表中的每一项都收集自其支持者，并依其个人经验赋以权重。而这个最终列表能够在多大程度上被采纳，事关自我价值认同和个人声誉。互投赞成票在小范围内很是流行——在这种特定的激励体制下是无可避免的后果。“我不给你添堵，你也别给我下绊儿。”

谁在需求制订过程中为产品本身——为其概念完整性、高效性、经济性和健壮性——摇旗呐喊？通常情况下，没有任何人会做这件事。同样经常性地，只有一个架构师或是工程师会做这件事，但他们只会根据自己的品味和直觉给出意见，到目前为止还未能掌握以事实为依据来说服他人。[2] 这是因为在经典的、基于瀑布模型的产品过程中，需求是在设计开始之前就确定了的。

这样做的结果显然会是一个过分臃肿的需求集合，这会成为一大堆愿望列表拉杂凑合的结果，且配合之间全无约束制约。[3] 通常情况下，该列表中的项目既未排定优先级，也未赋以权重。委员会中人际关系错综复杂，连加权的动作都会引发激烈冲突，更不必说排什么优先级了。

最终，这个愿望列表和约束都不得不作出让步。在实践中，产品设计师们通常会根据各自的用户模型对官方需求作出自身的隐式加权评估。[4] 在多数情况下，若未能做好加权工作，往往会造成设计师和深度用户以及若干需求制定者确实拥有的领域知识之间的脱钩。

若是以委员会方式制订需求，不免由它的本性所驱使，有着将产品研发过度的趋势（底特律汽车工业、过分臃肿的软件系统、名目繁多的苛捐杂税以及FBI使用的系统等）。也许委员会主导制订的规格说明书正是体积庞大、野心勃勃的软件系统极易遭到灭顶之灾的罪魁祸首。

在IBM OS/360的研发过程中，其需求是由来自市场部的一个大型委员会初步定稿的。身为项目经理，我不得不将这份需求文档断然拒绝，因为它完全不切实际，并且用以获取项目实质的、由架构师和市场一线人员以及实现人员组成的工作组的体积过小。

抵制需求膨胀和蠕变

需求激增现象必须予以钳制，方法是防患于未然或将这种势头扼制于摇篮之中。国家科学研究委员会下属空军研究委员会的一份颇具见地的报告中，就着意从这两个方面入手解决问题。

在首个阶段里程碑之前以及早期阶段的系统工程负责委员会的工作亦始自他们自己的恐怖往事。30年前的主流军事系统开发周期约为5年时间。现在，从项目启动到系统部署要花掉两三倍的时间，尽管技术进步和威胁来袭的速度都加快了。[5]

过去的项目之所以取得令人瞩目的成功，通常来说都有着一个或数个明确的首要目标以及进度紧迫性。这些项目一开始时只有若干个最为概要的需求。随着开发的推进，这些需求被精化成更具体的子需求以及主要性能参数，这都要仰仗鞠躬尽瘁、能力过人的经理们在系统功能和进度及成本之间持续做出平衡。

对于需求蠕变而言，无论是来自用户还是内部组员的压力，进度的紧迫性在过去往往是最好的挡箭牌。（在系统构建过程中，这也是我本人最好的挡箭牌。）不过该委员会观察到，国防部的采购已经不再像过去那样事事催逼，取而代之的是不断增加的“监管”机制层级以避免错误。这种趋势在技术型公司里也已是屡见不鲜了。

该委员会建议，即使在为新系统进行的重点技术开发开始之前，组织良好的系统工程工作就应该开始着手进行了。不过，他们并不建议初始需求规格在首个阶段里程碑之前就完成，而是在系统开发的过程中，在首个和第二个阶段里程碑之间，完成详细需求之初始表达的定义。

对于首个阶段里程碑中明确的主要性能参数以及第二个阶段里程碑中的清晰需求的定义中，那些在初始阶段的运营考验中仍能保持稳定的部分，对于高效的开发阶段而言乃是至关重要的。[6]

将需求蠕变作为头等大事来抓，是对付它的最有效办法。该委员会给出的第一优先的建议是：及早任命手腕强硬、经验丰富、领域知识到位的经理，并要求他们在整个初始系统交付期间全程参与。尔后，授权他们**“以其认为必要的方式度身定制标准流程和步骤”**。[7]

他们还敦促应用需求追踪矩阵以确保每一个被精化、定义和列出的需求都的确是从一个或多个初始的总体需求中派生出来的——确保它不是从某个用户代表的要求或是设计师的愿望出发而被悄悄混入的，其目的则是保证结果炉火纯青、推陈出新并深孚众望。[8]

罪念

假定：

- 有这么一个客户，他从来不索求无度，并且非常乐意为他的架构师和建筑工人的专业技能和辛勤劳动支付合理的价钱（也许是出于自身利益考虑，因为将来可能还用得着他们）。
- 他聘请了一位视己为客户代理人的架构师，渴望用自己的才华和专业技能帮助客户发现其真正旨趣之所在，并竭诚服务。
- 他的承包方充分理解并不折不扣地按照其要求做事，并在预算和工期范围内依照最佳性价比生产高质量的产品。
- 所有的项目成员都是诚实、本分的，并且他们之间的沟通非常到位。

那么：我就认为，

- 成本加成式的付款安排将给客户付出的每一美元带来最大的价值。
- 设计-构建法将是构建一个项目最快速的途径。
- 显式的螺旋模型过程（第5章将详细描述）将能够产出最符合客户需求的产品。

如果最后一点成立，那么我们又如何解释瀑布模型的生命力这样坚挺，尤其是在螺旋模型和合作进化模型已经赢得了更大的用户忠诚度达四分之一个世纪之久的前提下仍然如此？

回答就一个词：罪念：骄傲、贪婪以及惰性。我们都明白，上面这些假定其实是理想化的。读者们也许在阅读它们的时候会嗤之以鼻："所有这些条件同时成立的可能性简直为零！"因为人类是堕落了，所以我们无法信任彼此的动机。同样因为人类的堕落，我们也无法保证沟通的到位。

合同

基于这些理由，"写下来。"我们需要书面协议来在沟通过程中作澄清；我们需要具有强制执行力的合同来保护自己不受他人之不当行为或自身所经受之诱惑的影响。当项目执行人是由多人构成的组织，而并非只是个人的时候，一份详尽的、具有强制执行力的合同就尤为重要了。组织通常会比其任何成员都表现得更糟糕。

显然，无论是在某个组织内部还是多个组织之间，迫使目标、需求和约束在过早的阶段就确定下来是合同的要素。每个人都清楚一个事实，那就是所有（在合同里写的）的事项在晚些时候肯定会发生变化。（这将为不良行为打开方便之门："在合同上先让一步，等到有需求变更的时候再狠狠地抬价。"）所以，貌似是合同的这种要素最好地解释了何以瀑布模型会在设计和构建复杂系统时会这样持久地存在。

一种合同模型

那种要求完整的、所有方面都一致同意的一组需求的压力，说到底来自人的欲望——通常是来自机构的要求——比如说想达成一份固定价格合同，或者想保证某种特定的提交内容。由于这种要求和铁的事实可谓背道而驰——一如我们在第3章所讨论的那样——想要给任何复杂系统通过指定的方式完成一组完备的、精确的需求实质上是不可能的，除非是通过设计过程中的迭代式交互才有可能完成。

那么，那种百年老建筑的设计准则又是如何处理这个困惑呢？从根本上讲，这是通过一种截然不同的合同模型实现的。考虑常见的建筑设计过程：

- 客户为建筑开发一个方案，而非撰写一个规格说明书。
- 他和建筑师签承包合同，通常按小时或完成百分比计费，用来购买他的服务，而非指定的产品。
- 建筑师从他的客户、用户以及其他的利益攸关者处探访而得出一个更完备的方案，而这个方案仍然不打算用作一个僵化的能在合同上写成白纸黑字的产品规格说明书。
- 建筑师完成一个概念设计，用以估算方案的平衡点以及在预算、工期和建筑规范等诸方面的约束。这可以作为首个原型，可以让各个利益攸关者能够对它在概念层面上进行验证评估。
- 经过几次迭代以后，建筑师就会着手进行设计研发，通常这一步会产生更加详细的图纸、3D的缩小比例模型、实物模型，诸如此类。再经过利益攸关者方面的若干次迭代，建筑师将做出施工图纸和规格说明书。
- 客户采用这些图纸和规格说明书，为最终产品签订固定价格合同。

请注意这种长期演化模型是如何把关于设计的合同和关于施工的合同分离开来的。就算这两者是在同一组织内实施的，这种分离也会把很多事情弄得一清二楚。

当然，这个模型也并非是完全按部就班的。正如任何一个参与过哪怕是一个很普通的建筑工程的人都知道的那样，实际的施工问题以及后期无论是由于针对需要或是设计的评估引发的客户变更都会导致设计变更，这反过来又会迫使合同变更成为必要的工作了。

以上概述的经典建筑过程有它自身的不足之处，尤其是它会带来延期。下列条件中的建设工程：

- 客户–建筑师–承包商之间存在紧密的信任关系，
- 设计中的问题都是众所周知的，或者
- 工期很紧并且压力较大，所以风险较高也是可以理解的。

其正常流程往往被整合为并行执行的、流水线式的设计–构建过程。建筑师会重新组织他们的工作，所以设计图纸中承包商首先需要的部分会被首先做出来：这包括需要大的提前量的钢材、现场作业、地基。

那些需要满足逐项列出条件的系统工程，也类似地需要能够在设计-构建过程的基础之上进行。这里的主要挑战在于让计算机和软件制造商们识别出构造的顺序以及哪些组件需要大的提前量。

这个过程包括大量艰苦的思维工作。我力邀设计界积极参与到这场对话中来。[9, 10]

注释

1. 维基百科（2002-2009）的词条“RAH-66 卡曼契”描述了该项目的历史。它对于该直升飞机的参数描述印证了我对于规定需求的记忆：

卡曼契直升飞机配有极其精密的探测及导航系统，目的是要使它能够在夜间和恶劣天气下运行。它的机身被设计为能够比阿帕奇直升飞机更适合装入运输机或登陆运输舰，这就使得它能够更快速地部署到重要据点。如果运输设备不可用，卡曼契直升飞机高达1 260海里（合2 330千米或1 553英里）的转场航程甚至能够使得它独自飞赴海外战区成为可能。

在这个事件中，LHX从轻型武装直升飞机演化成了侦察直升飞机，卡曼契军用直升飞机的样子如卷首插图所示。在仅仅制造了两架飞机以后，整个项目被取消，因为无人驾驶飞机已经接手了侦察功能。（根据维基百科同一词条，卡曼契军用直升飞机制造了不止两架：美国陆军计划采购大约1 300架卡曼契编入侦查或轻装攻击部队，第一架于2004年引进美国陆军服役。由两架DEM/VAL（展示与验证阶段）原型机进行飞航检定测试与评估。其中的首架原型机，在1995年五月由塞考斯基航空公司（Sikorsky Aircraft Corporation）的直升机制造厂建造完成，并于同年12月进行它的首次试飞。两架原型机并在1998年进行了一整年的飞行测试。基于 DEM/VAL 阶段任务的顺利完成，整个计划因此进入工程制造与开发（Engineering Manufacturing and Development，EMD）阶段。在EMD期间，又建造了另外八架卡曼契，并在2006年6月时试飞。——译者注）

2. Squires(1986)，《The Tender Ship》，政府对于创新技术的收购研究。“一个贯穿全书的主题是：成功的关键在于让设计师成为产品的设计完整性的忠实支持者。”（玛丽·肖，来自评论家的评论）Squires敦促设计师们对于产品完整性要有高度的热情：

一个应用科学家或工程师对于设计目标显示出终极的忠诚，从它概念产生的一瞬到其最终建成投产，矢志不渝。

3. 一名匿名审稿人正确地指出，对于一位利益攸关者而言可能不过是银样蜡枪头的需求，但对于另一位来说就可能是十分要紧的。一方面无论如何，我看到的结果是，每种特定的产品特性都有其拥趸。另一方面，虽然任何人都想要高效率、小规模、高可靠性、高易用性，但这些需求在需求确定过程中却无人拥护，这主要是因为这些好处对于特定的产品特性会带来的影响无法在早期就了解。

4. 第9章讨论了设计师的用户心理模型。

5. 空军研究委员会(2008)，《Pre-Milestone A and Early-Phase Systems Engineering》。

6. 空军研究委员会(2008)，《Pre-Milestone A and Early-Phase Systems Engineering》，4。不过需要参见第50页，以下这段话可能会被误解：

一个人必须在首个里程碑之际明确地制订一组完备的、稳定的系统级需求和产品。需求蠕变的确是个必须处理的问题，保持一定程度上的需求灵活性也是有必要的，因为涉及可行性和实用性方面的教训是有的……当然，控制是必要的，但不是绝对的管死。

在与我的个人通信中，无论是该委员会的主席保罗·卡明斯基博士，还是美国国家科学研究委员会主任委员James Garcia先生，都和我澄清，该委员会的本意是第4页的那段话所表述的内容。Garcia先生如是说：

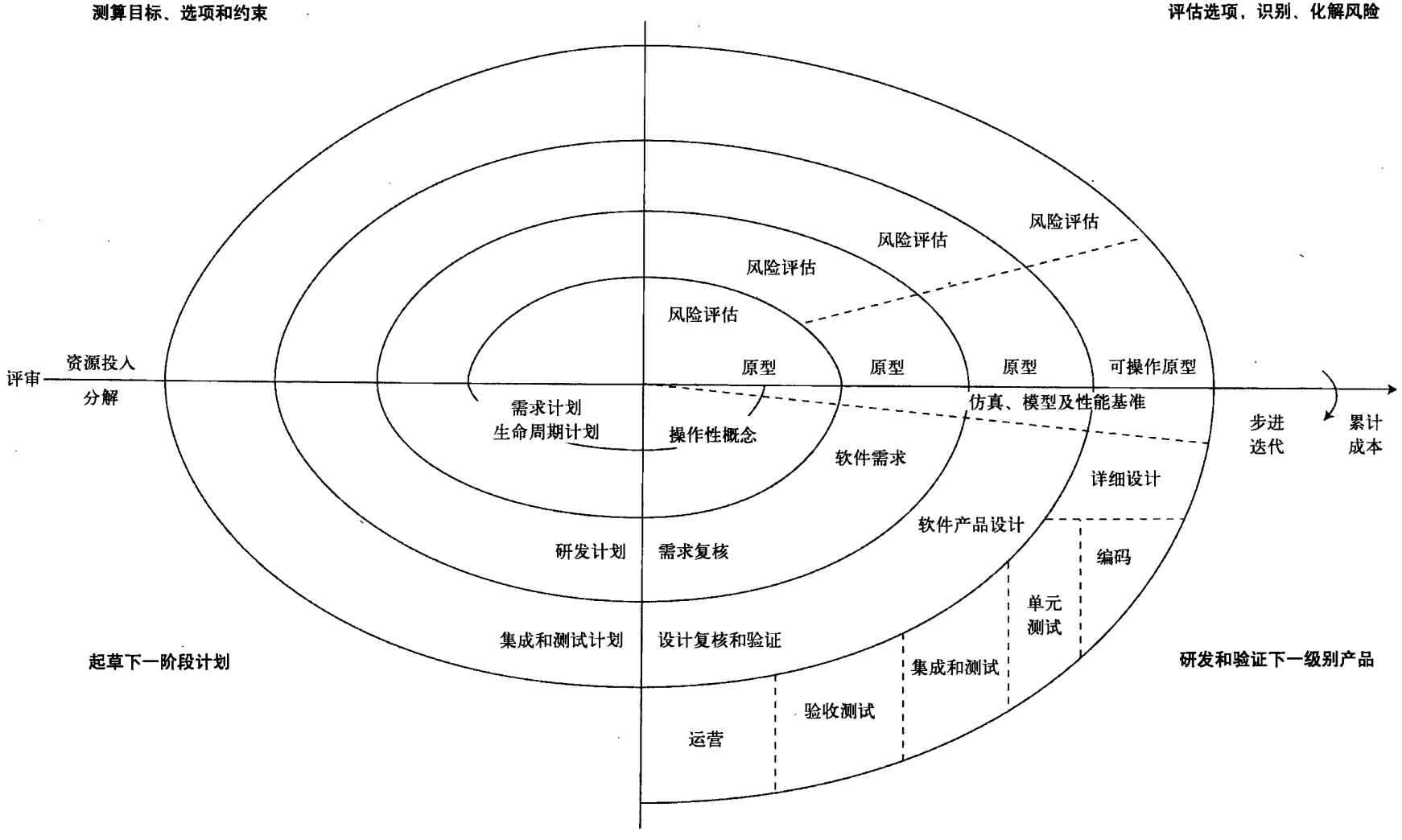
委员会的本意是说，明确的关键业绩参数（key performance parameters，KPP）在首个阶段里程碑中确定，在第二个阶段里程碑中确定明确而完备的需求，如摘要章和第4章中所述。本委员会认为，欲得到一个明确的、完整的系统级需求，在首个和第二个阶段里程碑之间与潜在承包商之间进行交互是不可或缺的。

7. John McManus，英国计算机学会院士，是项目管理和软件开发方法学的大师级实践者。Trevor Wood-Harper博士是斯坦福大学系统工程学教授。“能够展示出对于新方案的宗旨和期望的章程，以及项目经理对于未来工作任务安排的设想，对于IT项目来说乃是至关重要的出发点。”（McManus(2003)，《Information Systems Project》）

8. Boehm(1984)，“原型与规定”描述了一个课堂实验。在实验中，一个团队从精致制作的设计规格说明书出发，而另一个则实质上直接从需求条目出发，以便进行一个建筑活动。第一个小组饱受特性膨胀之苦，因为设计师不断向设计方案中塞入各种东西以“完善”它，或保持设计逻辑的前后一致。因此，并非只是列出要求的人导致了需求膨胀——设计师自己也在做这种勾当。我自己就不例外，在IBM Stretch计算机项目上就这样做过。

9. Jupp(2007)讨论了在英国市政工程中的公私合伙关系里，这种与众不同的合同方案的应用。

10. Muir Wood(2007)，“风险管理战略”是一份会议纪要，它就管道工程的客户和承包商之间如何处理其合同中不可预知的风险提出了建议方案。


测算目标、选项和约束
评估选项，识别、化解风险
风险评估
风险评估
风险评估
风险评估
评审
资源投入
分解
原型
原型
原型
可操作原型
需求计划
生命周期计划
操作性概念
仿真、模型及性能基准
步进
迭代
累计
成本
软件需求
详细设计
软件产品设计
研发计划
需求复核
编码
单元
测试
集成和测试计划
设计复核和验证
集成和测试
验收测试
运营
起草下一阶段计划
研发和验证下一级别产品

Boehm的螺旋模型

第5章

有哪些更好的设计过程模型

一种被广泛认同的观念是：创新性设计并不是先把问题定死，再去寻找一个令人满意的概念解决方案这么一回事；它似乎更像是对于问题的构造本身以及解决方案的思路这两者同时进行研发和完善的工作，这里面包括不断地在两个“空间”——问题以及解——之间进行循环往复的分析、综合，以及评估过程的迭代。

——Nigel Cross和Kees Dorst (1999)，“创造性设计中问题和解空间的共同演化”

为什么要有一个占主导地位的模型

无论是设计师领域的实践者还是教育家，现在都十分迫切地需要知道以下问题的答案：

- 如果理性模型真的是错的，
- 如果选用错误的模型真的是很要命，并且
- 这个错误的模型积重难返有其深层次的原因，

那么，有哪些更好的模型能够做到：

- 强调设计需求的递进式探索和演化，
- 能够被令人印象深刻地可视化，从而使得它们可以被团队和利益攸关者很容易地接受和理解，并且，
- 仍然可以在堕落的人类之间促进合同的达成？

既然是模型，顾名思义，它就是现实的简化抽象。因此，在设计的整个生命周期进程中会有很多种有用的模型，每一种模型都强调了一些方面，而略去了另外一些方面。Mike Pique制作了一个视频，戏剧化地强调了一点。他演示了用以表示蛋白质中牛血超氧化物歧化酶的约40种不同的计算机图形化模型：棒状模型、彩带模型、固体模型、反应模型，等等。[1]

这么一来，人们可以令人信服地辩称，寻求一个占主导地位的设计过程模型是傻瓜之举。为何不能让百家争鸣、百花齐放？每种模型不是都会有所贡献吗？

我坚决反对这种说法。在软件工程领域中瀑布模型的普遍存在——先不谈对它的诸多批评

和由于它的过分简化而造成的损害——使我明白了一个道理，那就是沟通的需求，以及学术指导的本性都意味着一个占主导地位的设计过程模型迟早会出现。总之，现在迫切需要的是采用一个具有更少误导性的模型作为替代品，而非只是为当前实践的现状再画蛇添足地弄出一个什么新模型。事实上，在更广义的设计领域，在我看来，Simon的问题解决模型事实上会导致大量企图理解或改进设计的人在死胡同里浪费大量的精力。

共同演化模型

Maher、Poon和Boulanger提出了一个正规的模型，我认为很有意义，这就是共同演化模型。[2,3] Cross和Dorst将该模型描述如下：

看来，创新设计并不是先把问题固定下来，然后再寻找一个满意的观念解法这么一回事儿。创新设计看起来更多的是将构造问题本身，以及寻找解法的思路这两者的研发和完善过程齐头并进，辅以分析、综合和评估过程在两个想象中的设计“空间”——问题空间和解空间——之间地不断迭代。Maher等（1996）提出的创新设计模型是基于设计过程中的问题空间和解空间的“共同演化”：问题空间和解空间是共同演化的，在此过程中信息在这两个空间之间流动（见图5-1）。[4]

演化一词，在这里使用并不是十分精确。只要是对于问题的理解，以及解法的研发在一点点地生成与评估，那么我们就说这个模型是演化模型。

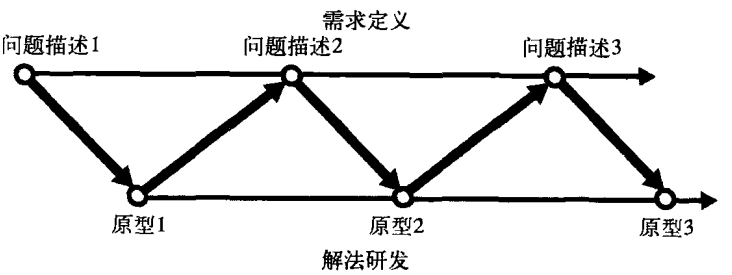


图5-1　Maher、Poon和Boulanger有关设计的共同演化模型（Maher (1996)）

技术哲学家们最近在深入研究这个问题：人类的创新活动能在多大程度上使用生物学进化论来建模。Ziman (2000)，一个由跨越多个学科的作者写就的著作，给出了21世纪初哲学家以其另外一些人在这个主题上的思想状况的一个很好的小结。[5] 它阐述了两方面的言论，即一方面生物学进化论是一个好的模型，由于它包括了随机迭代和自然选择；另一方面它又不是个好的理论，因为创新活动实际上是被有目的的设计所引导的（作者们假定进化过程并非如此）。

共同演化模型当然强调了对于需求的递进式探索和构造。它的视觉表现令人难忘。它并不包罗万象：它没有伪装成将设计-构建-测试-部署-维护-扩展这些过程的所有方面都包含在内的样子。此外，该模型的几何图像也没有示意一个收敛的过程。截至目前，该模型尚未进行大量的后续研发工作，而且在它原始的构造形态中，并没有表示阶段里程碑和合同的节点。虽然该模型有它的闪光点，也比理性模型要好，但我并不认为它已经充分完整。

旋转木马模型。然而，重要的是不要走火入魔。有一些提出来的模型，尽管丰富多彩，但却失之过分繁复，以致阻碍了理解，这样在记忆以及在讨论中使用起来机会就小得多了。例如Hickling的旋转木马模型，它带有很多圆圈，还有大圆里面套着小圆的结构。[6]

Raymond的集市模型

Raymond 在他的天才著作《The Cathedral and the Bazaar》（2001）中提出了一个观点：像建造大教堂那样的设计过程背后的整个观念都已经过时了，取而代之的是开源的——亦即“集市般的”——过程，它顺顺当当地，而又卓有成效地引领了Linux——一个功能强大而又安全可靠的操作系统的开发工作。他的论点非常鲜明，表述也淋漓尽致。他提出，开源过程可能是开发各类应用程序的最有效途径，开发操作系统也一样。他以一个自己的作品Fetchmail为例来解释他的观点。[7]

运作机理

Raymond 在他的集市过程中如是说：在用户与生产者社区中，某个成员发现了一个需要，然后开发出一个模块来满足它，并将该模块作为礼物提供给他的同伴使用。而这个将模块整合的过程，对于Linux社区而言，极大地受益于Linux的模块化结构，特别是它的管道和过滤器机制。同样的过程也适用于缺陷修复。某个成员在他所使用的模块中检测出一个缺陷，然后把它修复以使自己能完成手头的工作，尔后他就把这个修复成果作为礼物提供给社区。

显然，人们编写新的模块，以及修复缺陷是为了使他们可以做到想做的事。但他们又为什么将自己的工作成果分发出来，尤其是这么一来必要的测试、文档撰写以及发布等，都要求人们去做大量的额外工作呢？[8] Raymond的答案，我觉得有几分道理，那就是，这样做带来的激励和回报就是在社区中的威望。[9]

经常地，多个模块以及对于同一个缺陷的多个修复会同时提供给社区。Raymond提出，这时市场机制（即使对于免费商品而言）会发挥作用。更好的工具、修复结果，会赢得更广泛的受众，而它的作者也相应地会赢得更高的威望。

这样，集市就逐渐地被很多电子化提供他们的数字产品的“供应商”填充。许多买家，以投票的方式，通过增加在全球化社区中以电子化形式表现的威望来对他们给予回报。

模型优势

这种礼物-威望文化真是个奇迹啊！这与愚蠢而无同情心的“为金钱而工作”以及“把我的知识产权发生的权益拿来”等心态真是有着天壤之别！这对于其他的社会活动而言也是多么有意义的新模型啊！

此外，Raymond令人信服地提出，通过集市化过程生产出来的系统产品，在技术上一般而言优于那些由大教堂式过程中生产的同样产品。首先，从进化的视角来看，市场机制起到了选择拥有最佳设计的模块的作用。其次，将一个新的模块同时放到数百个测试者那里，会更快地发现其中的缺陷，从而催生更加可靠的产品。最后，缺陷被修复得更好，因为市场化的选择机

制同样作用于缺陷修复过程。

综上所述，集市过程是作为这样一个全新模型来到人间：它同时作用于产品构建以及协同的过程，团队成员通过电子化手段彼此联络，天各一方、互不熟识。

我力邀我的读者们都去读一读Raymond的作品——不仅是读他的那些脍炙人口的部分，而是把他书中的其他章节都读一读。那里有许多真理、洞见和智慧。

什么时候可以采用集市模型

无论如何，我想对开源过程发表6个彼此独立的短小评论。它们是一场旷日持久的讨论的替代品，因为我并没有使用它的亲身经历。

- 集市模型的确是一种演化模型。较大的系统是通过增加组件的方式扩张的，每个组件满足一种由具有用户和设计师双重身份的人所发现的需要（如果你愿意也可以叫它需求）。
- 这种礼物-威望经济学只能在有着其他收入以养活自己的人们中间才行得通；换言之，给到社区使用的作为礼物的软件，其实是其他工作所生产出来的产品所带来的副产品，前者才产生收入以支付建造者-捐献者所开支的费用。
- 由于许多这样生产出来的产品实际上是副产品，工具的数量就远比应用程序要多。产品也并不总是精雕细琢的，或经过充分调试的——它只要能够足以达成构建者所想要的目标即可。“市场”选择机制才是事实上的质量控制。
- 尽管有关开源过程的“开放”和“自由”的文字早已是铺天盖地，但是整幢Linux大厦的建成却很难看做是只砖片瓦的随机堆砌——Linus Torvalds始终作为一个保持其概念完整性的关键首脑力量存在。此外，对于Linux来说，功能规格说明早已有了：这就是UNIX。同样重要地，总体系统设计也是现成的。
- 所有设计过程都共有的关键要素在于对用户的需要、希望和验收标准的发掘。用于Linux社区的集市模型所取得的突出成功，在我看来是构建者同时也是用户这个事实所导致的直接结果。他们的需求来自于他们自己和他们所从事的工作。他们所要求的必要条件、验收标准和鉴赏品味则本能地来自他们自身的经验。整个需求的决定是暗中完成的，因而也充满了只可意会的技巧。我强烈怀疑，如果构建者自己并不是用户，并且对于用户的需要只有间接经验，那么开源模型还能不能行得通。
- 因此，大教堂式过程还是有市场的：仔细地做好架构，严格地控制过程，并且精心地做好测试。你难道会在构建新的国家航运控制系统时采用开源过程吗？[10]

Boehm的螺旋模型

Barry Boehm于1998年为软件系统的构建而提出了螺旋模型。[11] 本章的卷首插图给出了模型被提出时的原始形态。螺旋的形状当然表示的是过程。它将同一活动的连续反复彼此关联起来。这种几何形状很容易理解，而且令人印象深刻。该模型强调的是原型方法，它主张远在一个可以跑起来的原型成为可能之前，就从用户界面原型和用户测试起步。

该模型已经被广泛接受，它甚至被美国国防部的采购部门认可为瀑布模型的替代品。[12] 它也经过了一些研发工作。

Denning和Dargan（1996）对螺旋模型有如下批评："（这的确是一个进步，）但是它仍然是以设计师和产品为中心，而非以用户和行为为中心。"[13] 他们接着提出了一个相当非正式的以行为为中心的设计过程，其中恰当地给予了用户及用户模型以更多的关注。我很推荐他们这篇考虑周到的论文。

尽管如此，由于开发模型主要是供开发工程师使用的，我相信把它做成以设计师为中心也完全是合情合理的。此外，他们实现过程的这种尝试也没能照顾到要有一个令人印象深刻的几何模型的需要，也没法作为合同达成的基础。对于Boehm，以及作为其反对派的Denning和Dargan所提出的模型，我主张与具有代表性的用户保持经常性的而非连续不断的互动，沟通的手段则是相继改进的原型。

在螺旋模型的原始提案中，对于需求的渐进式探索有其位置但并未被强调，它也未能强调合同达成的节点。

我坚信，未来之路就是采纳和继续研发螺旋模型。我会建议通过以下的手段来加强模型中的螺旋：显式标注合同达成点；增加有关什么可以写入合同的清晰规格说明；有哪些是必然事件；有哪些是明显的风险所在。风险管理是Boehm之后大部分工作的重点。[14]

设计过程模型：第2～5章的讨论小结

- 一个正式的设计过程模型是必需的，目的是帮助组织设计工作、辅助为项目内部以及项目相关的沟通工作，亦有益于教学。
- 给予设计过程模式以可视化的几何表示至关重要，因为设计师们都擅长空间思维。他们最乐于学习、思考、分享和讨论有着明确几何图像的模型。
- 对工程师来说，设计的理性模型是自然而然的。的确，它被多次独立地、正式地阐述过，例如Simon、Pahl和Beitz以及Royce都有相关论述。
- 线性的按部就班的理性模型具有很大的误导性。它并未能反映出设计师的真实工作，或是一流的设计思想家所认定的设计过程的本质。
- 坏的模型流毒甚广。它会导致需求的过早固定，从而导致过度膨胀的产品，以及日程表、预算和性能的灾难。
- 理性模型在实践中积重难返，尽管它有种种不足，而且对它早有大量有说服力的批判。这是因为它具有诱人的逻辑简单性，也是因为构建者和客户之间需要"合同"。
- 已经提出过数个另外的过程模型。我发现Boehm的螺旋模型最有前途。我们必须持续对它进行研发。

注释

1. Pique (1982)，《What Does a Protein Look Like?》

2. Maher (1996)，“将设计探索正规化为共同演化过程”。

3. Maher (2003)，“作为设计的计算和认知模型的共同演化过程”。

4. Cross和Dorst (1999)，“创新设计中问题空间和解空间的共同演化”；Dorst和Cross (2001)，“设计过程中的创新思维”。

5. Ziman (2000)，《Technological Innovation as an Evolutionary Process》。

6. Hickling (1982)，“超越线性迭代过程？”

7. Raymond (2001)，《The Cathedral and the Bazaar》，我怀疑他想到“大教堂”一词是由《人月神话》第4章的卷首插图所产生的联想。

8. Brooks (1975)，《人月神话》，第2章。

9. Raymond (2001)，“金圣火盆”。

10. Richard P. Case睿智地评论道：

也许我们应该把开源设计中“所有人都可以更改它”的这方面和“所有人都可以感受到它”这方面区分开来。如果任何人都可以做改动，那么就没有人可以在某样东西（可能是至关重要的东西）出错时做出可信的分析，或给出有关它会被修复而不再会重现的任何保证。保密是另一回事。如果只有少数几个头脑了解实际的设计，那么缺陷就不会被及早发现，争鸣的概念也就不会被研发和评估（个人通信（2009））。

11. Boehm (1988)，“软件开发和改进的螺旋模型”。

12. 美国国防部自此明智地采纳了工业软件开发标准IEEE/EIA 12207.0、IEEE/EIA 12207.1和IEEE/EIA 12207.2。这些为螺旋模型提供了许可。

13. Denning和Dargan (1996)，“以行为为中心的设计”，110。

14. Selby (2007)，《Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research》，在第5章中收集了Boehm有关风险管理的最重要的数篇论文。

第二部分

协作与远程协作

Menn设计的Sunniberg大桥，1998

第6章

协作设计

开会是为了躲避“枯燥无味的劳作和孤独的思考”。

——Bernard Baruch，撰于Risen (1970)，“一种有关会议的理论”

协作在本质上是好的吗

自1900年以来，设计方面发生了两大变化：

- 绝大多数设计现在由团队完成，而不是由个人完成。
- 设计团队现在常常使用电子通信手段协作，而不是聚在一起。

这些转变导致的后果是设计社群热衷于讨论下面的热点话题：

- 远程协作（telecollaboration）
- 设计师构成的“虚拟团队”
- “虚拟设计工作室”

所有这些都是由电话、网络、计算机、图形显示和视频会议来提供支持的。

要理解远程协作，我们必须先理解协作在现代专业设计中的作用。

一般我们假定协作本身是一件“好事”。从幼儿园开始，“与别人玩得很好”就是一种高度赞扬。“我们在整体上比任何一个人都更聪明。”“参与设计的人越多，设计越好。”现在，这些有吸引力的陈述已经不再是不言而喻的了。我要说，这些说法肯定不是放之四海皆准的。

人类绝大多数的工作都是一个人完成的，或者是两个人紧密合作。对于19世纪和20世纪早期的大多数工程奇迹来说，都是这样的。但是现在，团队设计已经变成了现代标准，因为有一些很好的理由。危险就是产品丧失了概念完整性，这是一项非常严重的损失。所以现在的挑战是，在进行团队设计的同时实现概念完整性，而且实现协作的真正好处。

团队设计是现代标准

团队设计是现代产品的标准，对于大批量生产和一次性生产来说都是如此。这确实是自19世纪以来的巨大变革。我们知道那些18世纪和19世纪的杰出工程设计师的大名：卡特莱特（纺织机）、瓦特、斯蒂芬森（蒸汽机车）、布鲁内尔（桥梁、隧道、铁路、船舶）、爱迪生、福特、怀特兄弟。另一方面，请看“鹦鹉螺号”核潜艇（见图6-1）。我们知道瑞克沃（Rickover）是捍卫者，他使这艘核潜艇成为可能，但我们谁能说出主设计师的名字？它是一个技术团队的产物。

请考虑以下伟大的设计师，并想想他们的作品：

- 荷马、但丁、莎士比亚
- 巴赫、莫扎特、吉伯特和苏利文（Gilbert和Sullivan）
- 布鲁内莱斯基（Brunelleschi）、米开朗基罗
- 莱昂纳多、伦勃朗、委拉斯开兹（Vel á zquez）
- 菲狄亚斯（Phidias）、罗丹

大多数伟大的工作都是一个人完成的。偶尔有一些例外是由2个人完成的。2实际上是协作的神奇数字；婚姻是一种了不起的发明，有说不完的故事。

图6-1　“鹦鹉螺号”核潜艇

为什么工程设计从个人转向团队

技术复杂性。转向团队设计最明显的原因是工程每一个方面不断增加的复杂性。请比较第一座铁桥（见图6-2）和它壮观的后代（本章首页插画）。

第一座铁桥必须造得非常保守，也就是说，又重又浪费，虽然也很优雅。铁的特性和静态、动态压力的分布都没有得到很好的了解（虽然也了解得相当不错）。

Menn造的桥则完全不同，它以一种令人难以置信却又极为自信的方式屹立在那里，这是多年分析与建模的结果。

现代的实践中已经没有残留什么不成熟的技术了，我对这一点感触很深。我很荣幸参观了位于英国默西塞德郡阳光港（Port Sunlight）的联合利华研究实验室。我很吃惊地发现一名应用数学博士正在一台超级计算机上开展计算流体力学（CFD）工作，以便让洗发香波能正确地混合！他解释道，洗发香波是一种三层的乳剂，包含水性成分和油性成分，混合而不分离是很关键的步骤。

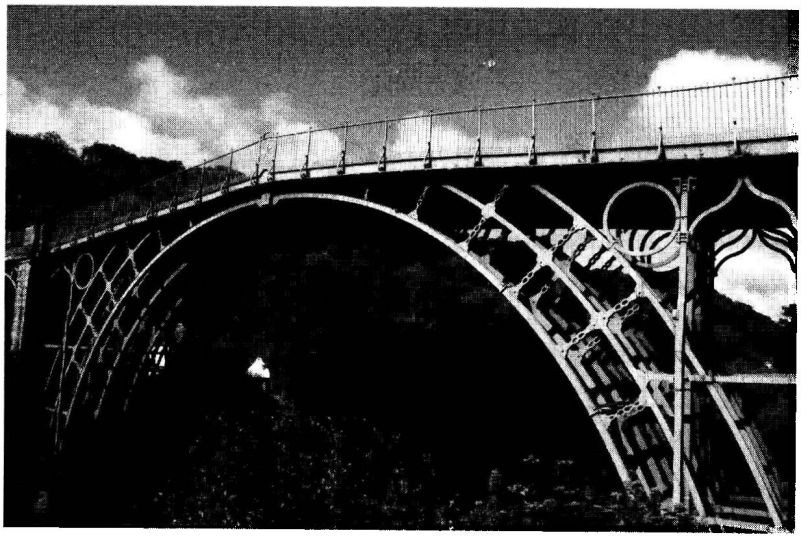


图6-2 Pritchard和Darby的铁桥，1779（英国施罗普郡）

约翰迪尔摘棉机的设计师利用CFD来安排气流，运送棉铃。现代农民不仅在拖拉机上工作数小时，也会在计算机前工作数小时，用于完成肥料、保护性化学物质、种子品种、土壤分析和作物轮种历史的合理搭配[1]。莎莉集团（Sara Lee）的大厨不断调整蛋糕的配方，以符合采购的面粉的化学特性；造纸厂的老板同样也在根据不同的造纸木浆的特点进行调整。

要掌握任何工程领域的爆炸性增长的复杂性，不得不依赖专业化。当我在1953年读研究生时，一个人就能追踪所有计算机科学的最新进展。当时只有两个年会和两份季刊。数学语言、数据库、操作系统、科学计算、软件工程，甚至还包括计算机架构（我的初恋），这些子领域的发展是爆炸式的。因此，我全部的智力生涯就是不断地放弃对这些令人激动的子领域的兴趣，因为我已不能追踪其进展。这类分裂发生在所有创造性的学科中，所以当今设计的最尖端作品需要一些专家的帮助，他们掌握不同的技艺。

这种对许多技术的详细窍门的爆炸式需求在一定程度上有所缓解，因为提供这种详细窍门的方式也令人吃惊地爆炸式增长：通过文档、有技能的人、分析软件，以及搜索引擎找到文档资料和可信的候选合作者。

快速推向市场。转向团队设计第二个主要原因是人们需要快速完成新设计、新产品，并推

向市场。经验法则表明，一类新产品中第一个推向市场的，可以合理地预期它会长期占据40%的市场份额，而其他份额由众多较小的竞争者来瓜分。而且，先行者可以在竞争的形成中收获丰厚的利润。在最大的胜利中，先行者保持着统治地位。这些事实对设计进度带来很大压力。团队设计因此成为必需，因为这样就能够在竞争环境中加快交付新产品[2]。

为什么这种竞争时间压力比以前更大？因为沟通的全球化和市场的全球化意味着，任何地方的好想法现在都会传播得更快。

协作的成本

“许多人手会让工作变得轻松。”——通常如此

但许多人手会让工作变得更多——总是如此

我们都知道第一句格言。这对于可以分割的任务来说确实是对的。每个人身上的工作负担变轻了，因此完成的时间就变短了。但没有哪种设计任务可以完美地分割，极少数的设计任务能较好地分割[3]。所以协作会带来额外的成本。

分割成本。分割设计任务本身就是一项多出来的任务。简明扼要并且准确地定义子任务之间的接口就意味着大量工作，还要冒很大的风险。随着设计进行，这些接口不断需要解释，不论它记述得多么准确。理解上会产生分歧。定义上会存在不一致的地方，解读时会发生冲突。这些问题必须调解。

为了简化制造，所有组件中的通用元件必须标准化，必须建立某种通用的设计风格。

然后，这些分离的部分必须集成在一起——这是对接口一致性的最终测试。这不是在船坞上的那种实际的集成：“计划时切开来，安装时猛敲”[4]。

学习与教授成本。如果n个人协作进行一项设计，每个人都必须了解目标、愿望、约束条件、效用函数等方面的最新进展。这个小组必须对这些问题持有共同的看法，即要设计的是什么。根据粗略估计，如果一个人的设计工作包含两个部分（学习了l和设计d），那么当这项工作由n人分担时，总体工作量不再是

$$\text{工作量} = l + d$$

而至少是

$$\text{工作量} = n\,l + d$$

而且，具有远见和知识的人必须教授其他人，因此在教授时不会进行设计。希望专业化带来效率的人必须承担这些成本。

设计中的沟通成本。在设计过程中，协作的设计师们必须确保他们的部件能够组装在一起。这要求他们之间有结构化的沟通。

变更控制。必须准备好一种变更控制机制，这样每个设计师所做的变更只有两种情况：

1）只影响他自己的部分；2）已经与影响到的其他部分的设计进行协商。因为设计的许多成本实际上是修改和返工，所以变更控制的成本很显著。没有正式变更控制的代价会更大。[5]

挑战是概念完整性

我们所认为的设计中的优雅实际上在很大程度上是指完整性，即概念的一致性。请看Wren的杰作——圣保罗大教堂（见图6-3）。

图6-3 Wren设计的圣保罗大教堂

如果工具中存在这样的设计一致性，那么它不只是让人喜爱，也会易于学习和使用。工具做了人们想要它做的事。我在《人月神话》中主张，概念完整性是系统设计中最重要的考虑[6]。有时候这一优点称为内聚（coherence），有时候称为一致性（consistency），有时候又称为风格统一（uniformity of style）。Blaauw和我在其他地方曾花了相当长的时间讨论过概念完整性，认为这是组件原则的正交性（orthogonality）、妥当性（propriety）和通用性（generality）[7]。单个设计师或艺术家通常是下意识地创造出具有这种完整性的作品，他们通常会在遇到问题时，以同样的方式做出一些小决定（除非有很特别的理由）。如果他不能创造出这种完整性，我们就认为作品有瑕疵，不是最好的。

今天，许多了不起的工程设计仍然主要是一个人或两个人的作品。请看Menn设计的桥梁[8]。请看Seymour Cray设计的计算机。他的设计天才源自于他个人对整个设计的全面把握，从架构

到线路、插件和散热，也源自于他能够在所有的设计领域里进行折衷考虑[9]。他花时间进行他能掌控的设计，虽然他使用并指导着一个团队。来自公司和外部的那些压力会让他的注意力从设计偏离到其他事务上，对此，Cray产生了强大的反作用力。他反复将他的设计团队请出他早期成功创建的实验室，他认为孤独比交互更有价值。他很自豪，CDC 6600的开发团队有35个人，“包括看门人”[10]。

人们看到这种模式（物理隔离、小团队、注意力高度集中和一个人领导）反复出现在真正有创意的产品设计中，而不是模仿的产品：例如，Joe Mitchell领导的Spitfire团队，在英国汉普郡雄伟的Hursley House里。洛克希德的Skunk工厂（Skunk Works）在Kelly Johnson的领导下工作，从那里诞生了U-2侦察机和F-117隐形战斗机。IBM在佛罗里达Boca Raton已关闭的实验室曾是IBM利用PC赶超Apple的大本营。

异议

并不是每个人都同意我主张的这个观点。一些人主张参与式设计的社会公正性，即用户有权力在他们使用的产品的设计中扮演重要角色[11]。虽然这种参与在建筑设计中是可行的（也是精明而公平的），但用户对大规模生产的产品的设计参与只能局限于预期用户的少数代表。这种意见必须考虑到取样的代表性，以及设计师的愿景。

其他人辩称我的论据是不对的，团队设计实际上一直都是平常的事情。[12]读者请自行判断。

如何在团队设计中获得概念完整性

任何产品，如果因为它很大，技术上很复杂，或要求时间很紧急，所以需要许多人共同设计，它仍然必须在概念上一致，符合一个用户的思维。[13]这种一致性通常是个人设计的自然结果，而协作设计实现这种一致性就是管理的功劳，需要投入大量的关注。那么，如何组织设计工作以实现概念完整性呢？

现代设计是各学科间的协商吗

许多作者（大多数是学院派）从今天的高度专业化中得出结论，设计的本质已经发生了变化：今天的设计必须以“多学科协商”的方式（在团队中）完成。虽然没有明说，但暗示已经很清楚，团队成员是对等的，每个人都必须满意。不是的！如果概念完整性是最终目标，对等成员的协商是膨胀的产品的经典做法！结果由委员会设计而成，其中没有人敢对其他人的建议说“不”。[14]

系统架构

在团队设计中，确保概念完整性最重要的方式就是授权给一名系统架构师。此人必须在相关的技术领域具有能力。他必须对要设计的这类系统拥有经验。最重要的是，他必须对系统的特点和目的具有清晰的愿景，必须真正关心系统的概念完整性。

在整个设计过程中，这位架构师是用户和所有其他利益相关人的代理、审批者和辩护人。

真正的用户常常不是购买者。这在军队采购中是很明显的事实，其中采购者（甚至是规格制定者）离用户很远。实际上，同样的系统可能有多个用户，在战略、营部和单兵等不同级别上使用它。采购者出现在设计桌上，坐在市场人员边上。工程师也在场。制造商也在场。只有架构师或建筑师代表着用户。而且，复杂的系统和简单的住宅一样，它的架构师或建筑师必须具有专业的技术优势，实现用户总体的长期的利益。这个角色具有挑战性。[15] 我曾在《人月神话》的第4～7章详细讨论过这个问题。

一名用户界面设计师

较大的系统不只需要一名主架构师，实际上需要一个架构师团队。所以架构完整性的挑战又出现了。甚至架构工作也必须划分、控制并重新集成。这里，概念完整性再次需要特别注意。

用户界面是对用户至关重要的系统部件，必须由一个人紧紧地控制。在某些团队中，主架构师可以完成这部分细节工作。请看MacDraw和MacPaint，早期的Mac工具实际上是由他们的设计师开发的。在大的架构团队中，主架构师的职责太大，不能亲自完成界面设计。无论如何，必须有一个人来做这件事。如果一名架构师不能掌握它，那么一名用户也不能掌握它。

这样一位界面设计师不仅需要大量的使用经验和聆听技巧，最重要的是，他需要有品位。有一次我问Kenneth Iverson（图灵奖获得者，APL编程语言的发明者），“为什么APL这么容易使用？”他的回答让人如读十年书：“它做你想让它做的事。”APL强调一致性，在细节上展现了正交性、妥当性和通用性。它也强调了吝啬，通过很少的概念提供了很多的功能。

有一次我参加了一个非常有雄心的新计算机系列的架构复查，即将来系列（Future Series），IBM的开发者希望它成为S/360系列的接班人。架构团队非常杰出，有经验而且有创意。我很高兴地听他们解释伟大的愿景。好点子太多了！在一个小时里，一名架构师介绍了强大的寻址和索引机制。在另一个小时里，另一名架构师继续介绍指令排序、循环和分支功能。另一名架构师介绍了丰富的操作指令集，包括对数据结构的强大的新操作。还有一名架构师介绍了内容丰富的I/O系统。

最后，我被压得透不过气来，问道：“你们能否让我跟懂得所有部分的架构师谈谈，以便了解一个大概？”“没有这样的人。没有人能完全理解它。”我那时就知道这个项目注定要失败——系统会被它自己的重量压垮。他们递给我800页的用户手册，我心里已经确信了系统的命运。哪个用户能掌握这样的编程接口呢？[16]

协作何时有帮助

在设计的某些方面，多名设计师实实在在地增加了价值。

确定利益相关人的需求和愿望

如果说决定设计什么是设计任务中最难的部分，那么协作在这个部分能提供帮助吗？答案

是肯定的！在研究未满足的需求或取代已有的系统时，一个小团队比单个人要好得多。通常，几个人会思考许多不同类型的不同问题。许多问题意味着许多没想到的答案。协作的团队必须确保每个成员都有充分的机会来探索他自己好奇的问题。

建立目标。在任何设计过程中，设计师开始都会与一些利益相关人谈话。这些谈话的内容是这项设计的目标和约束条件。最难的任务是弄清楚隐含的目标和约束条件，即利益相关人自己甚至都没有意识到的那些目标和约束条件。实际上，这些谈话（说了什么、怎么说的、没说什么）形成了设计师对效用函数的最初估计。

这个阶段的关键部分是观察用户今天完成工作的方式、使用的工具和所处的环境。用视频记录下这些观察，并一遍又一遍地观看，常常会有所帮助。

在这个阶段，让协作的设计师参与是极有用的。其他的人会：

- 问不同的问题；
- 发现不同的没有说过的事情；
- 对事情的说法有独立的或许是相反的看法；
- 观察工作的不同方面；
- 促进对视频的讨论。

概念探索——激进的可选方案

在设计过程的早期，设计师就开始探索解决方案——越早越好（只要没有人死心塌地爱上某个解决方案就行），因为提出的这些方案的正确性通常会诱发当时还未说出的用户需求或约束条件。

头脑风暴。这是进行头脑风暴的时候。设计团队的每个成员分别勾画出几个个人方案。团队成员再一起向彼此的方案提出激进的甚至是狂野的想法。这个阶段的标准规则是“关注数量”、“不要批评”、“鼓励发散思维”、“组合并改进想法”和“勾画出所有想法让大家能看到”。[17]更多的人意味着更多的想法。更多的人彼此刺激，可以产生更多的想法。

这些想法不一定更好。Dornbug(2007)报告了Sandia实验室的一项受控工业-规模实验：

对于产生电子设备想法的数量而言，个人至少和团队表现得一样好，不论是否采用头脑风暴过程。但是，当我们从三个维度（原创性、可行性和有效性）上判断其品质时，个人大大（$P<0.05$）超过了团队一起工作的表现。

竞争是另一种协作。在概念探索阶段，人们可以通过举行设计竞赛来利用并激发多名设计师的创造力。如果已知的约束和目标得到了具体的说明，同时仔细地去除了不必要的约束条件，这种方式就最有效。

几个世纪以来，这一实践已经在建筑设计中成为惯例。1419年，布鲁内莱斯基（Brunelleschi）通过赢得佛罗伦萨圣母百花大教堂（Santa Maria del Fiore cathedral）穹顶的设计竞赛而一举成名（见图6-4）。他的激进观念的可行性是通过按比例的模型来仿真的。这开启了新的景象，

在今天的圣保罗大教堂（St. Paul's）和美国国会大厦中仍能看见。

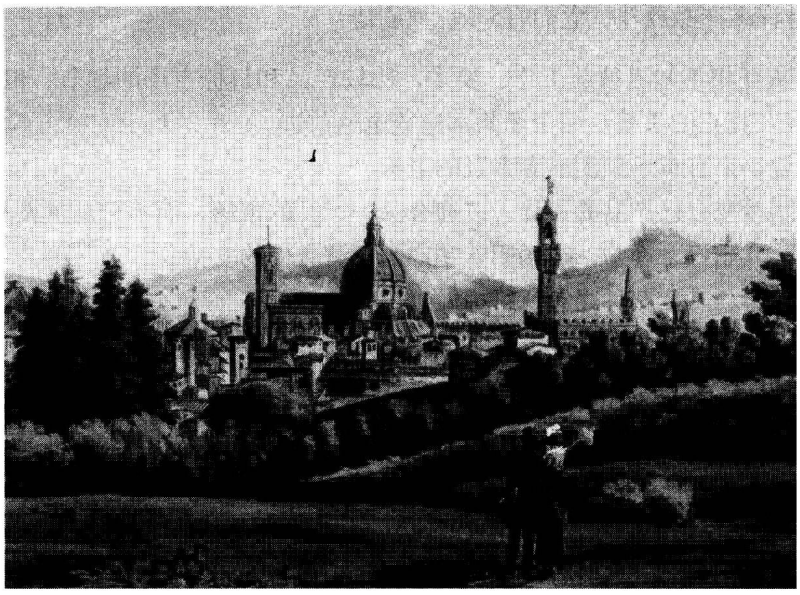


图6-4 布鲁内莱斯基的穹顶，圣母百花大教堂

在建筑和一些大型的城市工程工作中，只有一位客户，却有多位设计师希望得到这份工作。所以竞争就很自然地发生了。

在计算机或软件开发者的一般产品开发环境中，情况却相当不同。按惯例，一个团队被指定开发某个特定的产品。在团队中，总会有一些竞争的思想，反映一些不同的设计决定，争论也是常有的事。但很少有管理目的地建立多个团队，让他们为一个目标而竞争。

但是，在公司产品开发环境中，偶尔也会有正式的设计竞赛。在System/360架构的设计过程中，我们花了6个月的时间设计一个栈式架构。然后到了第一次成本预估时间。结果表明这种方式对中等以上的机器是适用的，但对于低端的7型系列来说，性价比不行。

所以我们进行了一次设计竞赛。架构团队自发组织成13个小团队（1～3人），每个小团队画一个设计草图，要满足一组固定的规则和最后期限。我作为裁判，认为13个设计中的2个是最好的。它们令人吃惊地相似，更令人吃惊的是这两个团队彼此非常冷淡，没有进行过沟通。

这些设计的融合奠定了项目的基础。（他们的最大区别是采用6位字节还是8位字节，这引起了整个设计过程中最激烈、最深入、最长久的讨论。）

这次设计竞赛最初是由Gene Amdahl建议的，我认为它带来了巨大的鼓舞和成果。它让每个人在士气受挫的成本估计之后又投入努力的工作中。它让每个人深入到设计的所有方面中，这极大地鼓舞了士气，在后来的设计开发中证实，这极有价值。它为许多设计决定提供了一致性。而且它得到了一个好设计[18]。

未计划的设计竞赛：产品战斗。设计团队B有时会太沉浸在它的设计中，从而与设计团队

A的市场目标发生重叠。这样就有了自然的设计竞赛，即产品战斗。

我曾看到过许多产品战斗。它们有5步标准剧情：

1）两个团队互相不知道彼此工作、要求、比较产品和目标市场的细节，但一致断定他们的产品之间不存在真正的重叠。两个团队都应该全速前进。

2）现实浮现，来自市场预测或者多疑的老板。

3）每个团队改变自己产品的设计，以迎合另一产品的市场，而不只是重叠的部分。

4）每个团队开始寻求客户、市场团队和产品预测者们的支持。

5）战斗发生了，直至某种强制执行力做出决定。

此时剧情发生了分歧：在竞争引起的紧张调查中，团队A胜利；或者团队B胜利；或者两者都生存下来；或者两者都没有生存下来。

多疑老板的早期行动可以缩短这种场景，通常也应该是这样。但是有时候，最好是让两种不同的设计方式走完（热情洋溢的）探索过程。

设计复查

协作最有价值的设计阶段就是设计复查，这甚至是必需的。必须复查多个科目：其他设计师、用户和（或）代理、实现者、购买者、制造者、维护者、可靠性专家、安全和环境监察员。

每个科目的专家必须单独复查设计文档，因为仔细的复查需要花时间反思，并可能需要研究参考资料、档案和其他设计[19]。每个人都会带来独特的观点；每个人都会提出不同的问题并发现不同的缺陷。但是联合的小组复查也是绝对必要的。

要求多科目的小组复查。小组复查有人数上的优势，但特别的力量来自于多个科目的观点。复查团队应该比设计团队大很多。那些需要按设计建造的人、需要维护它的人、用户代表、将要销售它的人——都必须包含在内。请考虑一种新型潜艇的复查。一位补给官员看到了一处缺点，他的担心触发了损伤控制专家的类似担心。制造工具专家看到了一些难以建造的部分，他建议的解决方案引起了声学专家的注意。

通用动力公司电船部门（Electric Boat Division of General Dynamics）的设计师告诉我在一次复查中，船坞工头看了一眼半圆筒状的存储水箱，马上建议卷出一件圆筒，切成两半，再在顶上封盖一块平板。这个部分按工程师的规格说明需要20个部件。那名工头说："我们造潜水艇的擅长卷圆筒。"

与此类似，位于英国莱瑟黑德（Leatherhead）的Brown & Root公司的一名设计师跟我讲了深海石油钻井平台的一次设计复查。维护工头指着某个部分说："这个部分最好用厚规格的钢。"

"为什么？"

“这样我们就可以在它安装之前，先在车间里刷好漆。当它安装到位后，我们就再也不能够刷漆了。”

工程师们重新设计了平台上这个部分的附近区域，让这个部分可以够得着。

利用图形展示。对于设计复查来说，最重要的辅助手段就是产品的普通模型。它可以是一张图、一个全尺寸的木头模型潜艇或虚拟现实的模拟潜艇、一个机械部分的原型，或者是一个计算机的架构图。

多科目设计复查通常要求设计的大量各种图形表示，这超出了设计师本身使用的表示方式。并非每个复查的人都能够从工程或架构图中想象出最终的产品。根据我对各种设施的观察，我发现这样的设计复查可能是虚拟环境可视化技术的最有效果的应用[20]。

分享产品模型和分享其他人的建议对于有效的设计复查都非常重要，当并非所有参与者都能现场出席时，模拟这种分享的工具就是小组设计复查的必要条件。这时电子协作就发挥作用了。

协作何时无用——对设计本身

对于设计协作的幻想式概念。在计算机支持的协作工作文献中，加入了对协作设计的幻想方式。这没有什么问题，只是这种欺骗性的概念更多关注复杂的学术研究，而较少关注对协作有用的技术工具。

在这种幻想中，设计团队真实看到设计对象的模型，不论是一座房屋、一个机械部件、一艘潜艇、一张软件的白板图或是一段共享的文字。所有团队成员都会建议修改，通常是直接在模型上进行修改。其他人建议修订，讨论继续下去，设计就一点一点地形成了。

并非协作设计的方式。但这种幻想式概念并不是协作完成设计的真正方式，它与设计复查不同。

在我看到的所有多人设计团队中，设计的每一部分在任何时候都有一个负责人。这个人负责准备这一部分的设计建议。然后他与协作者会面，这实际上是小型的设计复查。然后他退回去，根据协作讨论的决定和方向进行后续的详细工作。

如果在这个过程中提出了一些不同的建议，但负责人没有接受，那么建议者通常会撤回建议，并完成一个可选的设计方案。然后再次召集会议，选择方案、融合方案或转向第三种设计方案。

在哪里进行设计控制？幻想式概念不能够产生设计，只能够改进设计。作为协作式设计变更的模型，幻想式概念也是有缺陷的。从协作得到的进度计划意味着并发的活动，而并发的活动就意味着需要同步，这在个人设计中是完全不需要的。设计师Jack负责远洋航行中油轮中的空气管，Jill负责蒸汽管道。每个人完成他自己的设计，在后续的改动中，必须通过某种设计控制机制进行监控，保证他们没有占用相同的空间。必须准备好某种冲突解决过程来处理冲突。

必须建立某种版本控制机制，以保证每次针对早期设计的某个打上时间戳的版本所做的设计修改都能有效工作。

有一次，我看到这种幻想式概念确实被提出来，作为客户的海军上将查看了一艘核潜艇的设计模型，他移动了隔离壁，让设备维修者能更好地工作。（要做到这种程度，对CAD的虚拟现实接口是一项有技术挑战的任务。许多实时的虚拟技术依赖于大多数世界模型的静态特征。）

但这种挑战不值得接受！海军上将可能想移动隔离壁，了解一下空间看起会怎样，也许他可以在一个受保护的模型上做这事。但在这样的移动成为标准设计的一部分之前，必须由某人或某个程序来检查对隔离壁另一边的空间的影响、结构上的后果、声学上的后果、对管道和线路的影响。请想象一下一名负责任的工程师发现隔离壁被海军上将移动后的恐怖心情，上将不可能知道约束条件和由此导致的设计折衷。到了设计能让上将实际查看时，正式的变更控制要求已经实行很久了。

协作设计的幻想式模型反映了对概念完整性明显的不关心。Jill在这里拍拍，Jim在那里推推，Jack在那边打个补丁。这很自然，这是协作，但这得到的是糟糕的设计。实际上，我们对这个过程非常了解，所以我们给了它一个带有蔑视的称谓：委员会设计（committee design）。如果协作工具的设计目标是为了鼓励委员会设计，那么它们弊大于利。

概念设计尤其不应该协作

在完成了探索阶段，选定了基本主题之后，接下来就是概念完整性的统治时间。设计源自一位主设计师，由设计团队支持，而不是分解完成[21]。

这样追求的概念设计可能会走进死胡同。然后就必须选择另一种基本方案，在新的基本方案选择之后，又能够进行有序地协作探索。

两人团队很神奇

前面讨论的设计协作讲的是超过两个人的团队。两人团队是特例。即使在概念设计阶段，在概念完整性最宝贵的时候，齐心协力的结对设计师可以比单个设计师更有效。结对编程的文献表明，在详细设计的时候是这样的。通常开始的生产效率不如两个人独立工作，但错误率大幅降低[22]。既然许多设计中40%的工作是返工，那么实际生产效率就提高了，产品更加健壮。

这个世界充满了两人协作。木匠需要有人抬起横梁的另一端。电工要人帮忙让线穿过扣环。抚养小孩最好是由两个积极协作的父母来完成。“一个人孤独是不好的”，虽然这句话的实际意思是在讲婚姻[25]，但也适合向设计独行侠布道。

典型的两人设计协作机制看起来似乎与多人设计和单人设计不同。两人会快速地、非正式地交换思想，他们之间既没有协议规定谁有发言权，也没有规定谁受谁支配。每个人都有一段时间的发言权。这个过程在微型会议建议、复查和批评、反面建议、综合和决定之间快速切换。思想形成通常有一条主线，不用像多人讨论那样维护分离的思路。两支铅笔可能在同一张纸上

移动，没有冲突和矛盾。

“铁磨铁，磨出刃”，两人彼此激励，比单人设计时更积极。也许正是思考表达的需要（在说明是什么的时候同时说明为什么），才导致人们更快地理解自己的错误，并更快地意识到其他可行的设计方案。

Torrance在1970年的经典论文中表明，两人互动产生的思想是原来的两倍，原创性思想也是原来的两倍，同时也增加了快乐，导致实验对象去尝试更困难的任务[24]。

结对设计的对话仍然需要由一个人穿插起来——形成细节，记录下创造性的成果，并为下一次对话准备一些建议。

对于计算机科学家意味着什么

许多学院派计算机科学家努力设计一些计算机辅助协作的工具，由他们自己和其他学科的人使用。让人痛苦的是这些想法和工具很少进入日常生活。（成功的重要工具是代码控制系统和Word中的“修改痕迹保留”。）也许这是因为学院派工具制造者特别容易忽视真实团队设计的一些关键特点：

- 真实的设计总是比我们想象的更复杂[25]。由于我们常常从教科书的例子开始，这种例子必然是过于简化的，所以这一点就尤其正确。真实的设计有着更复杂的目标和更复杂的约束条件要满足，对于满意程度的测量也更复杂。真实的设计总是爆发出无数的细节。
- 真实的团队设计总是需要一个设计变更控制过程，以免我们左手弄坏右手创造的东西。
- 无论多少协作都不能消除对“枯燥无味的劳作和孤独的思考”的需要。

由于这些原因，我认为我们应该非常注意，避免为少有实际设计经验的研究生指定协作设计工具领域的论文题目。而且，在遇到并非基于真实经验和真实设计应用的论文时，我们的期刊应该非常小心，慎重接受。

注释

1. Economist (2009), “Harvest moon”。

2. 聪明地组织多个项目的经理会尽早让一名设计师或一对设计师开始探索可预见的技术，但这种技术现在还不能应用。

3. Brooks (1995),《人月神话》，第2章。

4. 电船公司的船坞工头——Groton，康涅狄格州（私人交流）。

5. 我所见到的关于单独设计师和个人设计师最完整的科学研究是Cross (1996a)的《Analysing Design Activity》。

代尔夫特会谈记录（Delft protocols）包括一名单独的设计师和一个3人团队来攻克同一问题，对双方都进行视频录像，鼓励单独的设计师大声说出他的想法。20个不同的小组，每个都使用自己的分析方法，以便分析代尔夫特视频会谈记录。大多数小组应用他们作者自己事先定

义的活动分类来分析一个或两个会谈记录。许多小组要么比较了两种不同方式的活动和表现，要么分析了团队的社会行为。最明确的结论是Gabriela Goldschmidt (1995)，在“The designer as a team of one”中做出的：“详细分析得出结论：个人和团队在得到工作成果的方式上几乎没有区别。”

Charles Eastman (1997) 在《Design Studies》(475～476)中对这本书进行了评价：

这些研究得出了一组丰富的观点，让读者能够理解设计过程中的丰富性和个人心理上的特点。视频录像显然记录了设计行为的丰富特点……但是，当前会谈记录分析方法的局限性也很明显。每组研究本身只是对整体设计过程的一孔之见。只有通过累积多组研究，完整过程的意义才能够展现出来。

经过了三十年的努力，这本书清楚地展示了设计会谈记录研究当前的状态，并将这些研究与各种设计理论更广泛地联系起来。

6. Brooks (1995)，《人月神话》，第4章，42。

7. Blaauw and Brooks (1997)，《Computer Architecture》，Section 1.4; Brooks (1995)，《The Mythical Man-Month》，Chapters 4-7, 19。

8. Billington (2003)，《The Art of Structural Design》，Chapter 6; Menn (1996)，“The Place of Aesthetics in Bridge Design”。

9. Blaauw和Brooks (1997)，《Computer Architecture》，第14章。

10. Murray (1997)，《The Supermen》。

11. Greenbaum and Kyng(1991)，《Design at Work》；Bodker (1987)，“A Utopian Experience”。

12. Weisberg (1986)，《Creativity: Genius and Other Myths》；Stillinger (1991)，《Multiple Authorship and the Myth of Solitary Genius》。

13. R. Joseph Mitchell是“喷火”（Spitfire）战斗机的设计师，他警告他的一名试飞员（用户！）要注意工程师：“如果有人告诉你关于飞机的什么事情，复杂得你难理解，听我一句话：那都是扯淡。”

14. Artechra 公司的Eoin Woods说：

我对联合设计不像你那样悲观。我曾在多个团队中工作过，在那里我们有触发灵感的讨论，推动着我们的设计，然后得到我们一致同意的解决方案（虽然有时候有一个仁慈独裁者做出最后的决定）。设计仍然是一致的，因为它有一两个强大的设计概念占据了统治地位，然后驱动着所有其他决定，我们不是在搞委员会设计，然后再在详细决定上讨价还价。

15. Brad Parkinson现在在斯坦佛，他是GPS系统的两位系统架构师与联络官员之一。他指出，对一些系统部件拥有多个签约方使得该项任务的挑战性大大增加。(私人交流(2007))。

16. 卡内基梅隆大学的Mary Shaw问道：“这对现代软件开发环境及其API有什么意义？”

17. Osborn (1963)，《Applied Imagination》。

18. 组织设计中的设计竞赛现在还不太一样，这种任务在本质上是政治的。各种竞争力量甚至常常没有共同的目标，即要让组织机构工作得最好。组织机构工作得多好要让位于谁有什么级别的权力。

19. 玛格丽特·撒切尔说：“人们希望得到文档（而不是视图演示），这样人们就能事先仔

细思考，并咨询同事。”（与John Fairclough爵士的私人交流）。美国商人太多时候是使用PowerPoint演示来进行复查。这些模糊的报告让每个参与者凭着兴趣自己解释这些信息，也有助于避免关键而令人尴尬的细节。

路易斯·郭士纳是实现IBM转型的CEO，他很早就让整个公司文化感到震惊。在（(2002)，《Who Says Elephants Can't Dance?》，43）中，他说：“Nick当时正在进行第二次演示，我走到桌子前面，在他的团队面前尽可能有礼貌地关掉了投影仪……这产生了强大的连带效应……我指的是惊慌失措。这就像美国的总统禁止在白宫会议上使用英语。”

20. Brooks (1999), “What's real about virtual reality?”

21. Harlan Mills的“得到支持的主设计师团队”的概念，即“外科手术式”团队，在Brooks (1995)，《人月神话》，第3章中有详细的介绍。

22. Williams (2000), “Strengthening the case for pair-programming”；Cockburn (2001), “The costs and benefits of pair programming”。

23. Torrance (1970), “Dyadic interaction as a facilitator of gifted performance”。

24. 例如，参见Hales(1991)的令人印象深刻的博士论文：“An analysis of the engineering design process in an industrial context” (Cambridge)，或参见Salton (1958)的“An automatic data processing system for public utility revenue accounting” (Harvard)，其中详细记录了实际设计中所涉及的工作。

Henry Fuchs对未来办公室的想象

第7章

远程协作

新的电子相互依赖关系让世界再现了地球村的图景。

——Marshall Mcluhan (1967),《The Medium is The Message》

为什么要远程协作

我们终于可以探讨远程协作了。为什么设计团队现在利用通信技术，让不在一起的人们能够协作呢？

专业化

现在技能的过度专业化导致了许多协作。但并非每个村庄都有所有特定的专业技能——甚至城市也是这样。曾经到处都是的乡村铁匠现在已变成了稀罕的钛合金材料科学专家，小镇工头已变成了Red Adair公司，被召唤到地球的各个角落去扑灭价值几百万美元的油井大火。[1]

家

对于住在哪里，人们有强烈的偏好，这种偏好甚至是最重要的。对于许多人来说，这意味着家、家族和文化的召唤。对于另一些人来说，这意味着乡村、小镇和城市的区别。对还有一些人来说，这意味着气候、海滨或山区的差异。具有高度专业技能的人通常可以自行决定。远程协作技术让越来越多的这种专家能够住在他们喜欢的地方，并在别处工作。在我自己以前的学生中，有一个住在冰岛，有一个住在巴西，而他们“在”硅谷工作。

整天工作不停

地球的自转让工作能够整天不停地推进，团队的各个成员交替地在白天工作。

成本

生活成本和生活标准的不一致，使得人们通过外包能够以极低的成本获得通用的高科技技能。当然，在不同的经济体之间挖掘一条（电信）通道引起了夷平世界的大潮，这肯定是最健康的“外国”援助方式。

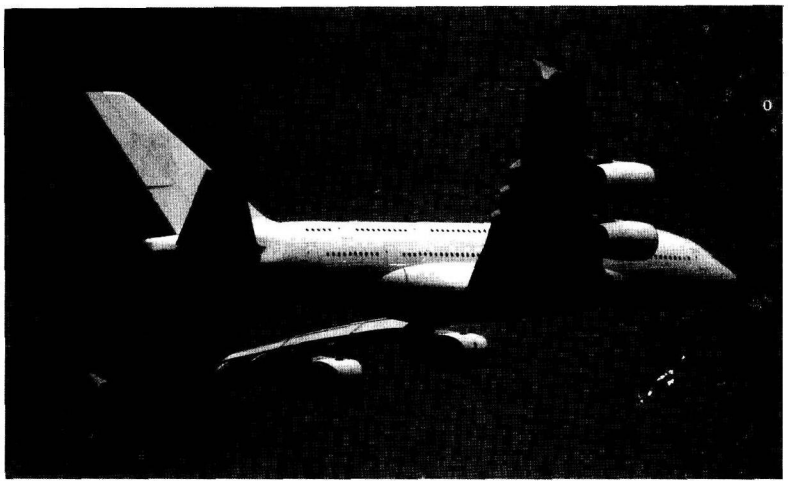


图7-1　空中客车A380

政策

政府支持的大型跨国公司不可避免地涉及在不同国家分派工作的问题，因此工作地点也不一样。请考虑空中客车380，这是一项大胆的工程工作（见图7-1）。不仅是制造，开发工作也是分别在法国、德国、英国和西班牙进行。

Jeffrey Jupp当时是空中客车英国的技术主管，他向我解释了空中客车的机翼是如何在布里斯托尔（Bristol）设计，又如何安装到在图卢兹（Toulouse）设计的机身上的：

- 全部使用了电子通信能力。
- 布里斯托尔将一些自己的工程师派驻到图卢兹作为代表。
- 每天都有一架公司的飞机往返于布里斯托尔和图卢兹，双向都载人。

根据我的经验，这些协作方式没有一项可以省略。可是，空客380还是暴露了一个特殊的缺陷，这个缺陷对于因政治而导致的分布式开发可能更具危害性。法国和英国的团队使用了CATIA CAD软件的第5版。德国和西班牙团队使用了第4版。你瞧，真想不到，一个团队设计的布线系统需要更大半径的导管，而另一个团队提供的导管却较小，部分原因正是这两个软件版本的区别。这批飞机和其他初始的交付延迟了大约22个月，成为一段痛苦的日子。[2]

到那里，做那事——IBM System/360计算机系列的分布式开发，1961～1965

最初的7台IBM System/360计算机是在3个国家的4个地点同时开发的：纽约州的波基普西（Poughkeepsie）、恩迪科特（Endicott）、英国的赫斯利（Hursley）和德国的波布林根（Böblingen）。这些计算机是第一批严格向上-向下二进制兼容的系列，在业界率先从6位字节转向8位字节。我是项目经理。本书第24章是System/360架构设计的案例学习。（Model 20不是向下兼容的，在我看来，这是架构师William Wright思考中所犯的一个错误。）

超过40种8位的输入/输出设备必须同时开发，每种设备都要用到特殊的技能和经验，这些技能和经验在分散得更广的实验室里：法国的戈德（La Gaude）、瑞典的利丁厄（Lidingö）、荷兰的厄伊特霍伦（Uithoorn）、加州的圣荷塞（San Jose）、科罗拉多州的波尔得（Boulder）、肯塔基州的列克星敦（Lexington）、纽约州的恩迪科特（Endicott）。技术创新极大地促进了这些工作的协调——精确定义的标准逻辑、电气和机械接口可以将任何I/O设备接到任何一台计算机上。[3]即使是这样，管理这些分布式的开发仍然是一项主要任务。软件开发的分布更广。

对于计算机、软件和I/O设备，我们使用的管理技术和前面介绍的英国航空航天系统公司（BAE）一样。我们的电信设备当时要原始得多：我租用了IBM第一条全天候翻译电话线。我们没有开一架公司飞机往返于各地，但我们买了很多机票。英国实验室保持有一名常驻参与者在Amdahl的波基普西架构小组；我们在英国和德国的处理器实现团队中，保持着来自波基普西的多名常驻参与者。

除了成千上万的电话和文档之外，许多双边的面对面会议将这些实验室联系在一起。每年举行持续两周的全部团队成员会议，解决悬而未决的冲突和挑战——每次会议解决大约200个这样的问题。

我们的分布式开发工作是因为那些往常一样的原因：

- 分布在各地的技术专家
- 不能移动的天才人员
- 分公司之间的政策和工作的划分

这些努力取得了极大的成功。[4]但是不要搞错：我们的工作是分布式开发一件统一的产品！而且，分布式开发实际上制造了许多的额外工作！我们低估了同一地点的团队工作中，非正式沟通渠道的巨大重要性，这给我们带来了痛苦，直到后来我们感觉到缺少这种沟通。空间障碍是真实的！[5]时区障碍是真实的，有时候比空间障碍更真实！文化障碍是非常真实的，必须加以考虑！[6]

让远程协作有效

分布式设计只会越来越多。通信技术继续爆炸式增长。设计师和设计经理如何利用通信技术来实现远程协作呢？

面对面的时间很重要

请想想你自己的电话对话。在与陌生人谈话时，除了感觉不舒服，你是否还感觉到效率上的差异？和与熟人谈话不一样？

在什么情况下你会亲自过去，避免利用视频会议、电话、电子邮件、书信等手段完成：

- 订午餐？
- 在购买服务时寻求折扣？

- 在复杂的商务交易中谈判?
- 计划家庭度假?
- 解聘你的管理助手?

有些情况下，你会选择电子邮件或电话，而不是走过去（而且在时间上同步）；在另一些情况下，你可能会很开心地走相当长一段距离。

我所知道的最成功的远程协作建立在大量的面对面沟通的历史基础之上，即使是这样，在通信沟通的过程中仍然需要一些面对面的时间。如果之前从未进行过面对面沟通，那么出差在金钱和时间上都是值得的。

我在IBM花得最值的一些钱是租了一辆巴士，将S/360项目的管理人员和秘书带到离波基普西60英里的怀特普莱恩斯（White Plains）。他们与分公司总部的对应人员共进午餐并交谈，声音很熟悉，但之前从未谋面。这种磨合比更大的合作压力有效得多。

有人告诉我，在设计开始时，波音公司把777型飞机的几十个分布式设计团队带到华盛顿的埃弗雷特（Everett），让他们在一起待上数周的时间。

人们直觉上知道面对面时间的价值。所以，尽管有强大的视频会议技术，飞机仍然运送着大量商务旅客。

干净的接口

在远程设计的组件之间定义干净的接口是很难的工作。这项工作不是定义好就结束了，事实证明，不断问答和解释定义的语义是必需的。必须进行变更、控制变更和充分沟通。

系统架构的另一个重点是不光要定义接口，而且管理层要设计一种预先确定机制，以解决观点或品味的差异。权威是不可替代的。

但是这些代价不菲的劳动的回报是难以置信的！干净的接口让设计中的错误率大为不同。有人曾估计，虽然错误和返工只影响到一小部分设计，但可能占到设计成本的一半。更糟糕的是，因为模糊或粗心的接口而产生的错误常常很晚才发现，即在系统集成的时候。它们更难发现，修复的成本更大，影响了整个系统的进度计划。

而且，干净的接口增加了工作的乐趣。设计是有趣的：解决同事间的误解通常不是有趣的。在设计时，人们感到取得进展；在解决接口误解时，人们感到时间在拖延。干净的接口让多位设计师感到拥有的快乐，在一件作品上有签名权的快乐。他们也有利于后续的拥有权，一些小的组件聚在一起，形成可辨识的更大的子系统。

远程协作的技术

十年复十年，技术权威预测设计师们的旅行将因通信技术进步而消失。这还没有发生。[7] 为什么？这会发生吗？我猜测越来越方便和逼真的通信技术将确确实实地取代越来越多的面对面会谈。[8] 但是，因为人类沟通中无穷的细微差异，经常坐在一间屋子里对设计协作者来说永

远是非常重要的事情。

低科技常常已经足够

文档。对远程协作来说，最有力的技术就是文档共享，不论是通过网络还是通过邮递。正式的文章和正式的图画带来了准确性，督促学习，鼓励批评，激发互动。

Gerry Blaauw和我发现，当我们写出1 200页的《Computer Architecture》(1997)时，我们大部分的横跨大西洋的交互实际上是通过邮寄手稿完成的。但是，这种有效的远程协作是基于9年的每天面对面的工作；基于由此而导致的我们对彼此的设计风格、敏感性和“协作方式”的深入了解；基于我们对计算机架构的共同深入的信念。即使是有这种基础，多次远程交换手稿仍然不够，必须辅以每季度的电话会议和每半年会面3天。

上面提到的这些事情总是有教益的。本质上，它们关注了还没有被解决的难题。我们发现当一段文本通不过时，总是因为我们还不知道我们在说什么。通常会接着进行半小时的讨论。我们又学到了关于计算机架构的新东西。

以前红笔标注的手稿，相当于现代的保留修改痕迹的Word文档。许多评论者可以互动，每个人的修改都可以与其他人的修改区分开来。Word的修改痕迹保留是一项设计得很好的功能。但我仍觉得红笔标注的文档更容易创建和研究，主要是因为它容易以两维的方式访问。我们的电子技术还做不到这一点（或者是我孤陋寡闻）。

电话。文档之后就是电话，这是比电子邮件更大的突破。电子邮件用户知道它的危害，它是仓促的文字，没有语音语调的变化，没有立即的反馈。即时消息是电话的糟糕替代品。

电话加上共享文档。电话加上共享文档比单独使用其中一项要强大得多。这种组合加上了实时交互，这节约了许多书面的解释，也消除了许多误解。不太容易注意的是，共享文档为电话交谈添加了许多规范和细节。必须逐字逐句地达成一致意见，这迫使大家共同面对许多问题，而在其他情况下这些问题可能会遗漏。

这种组合非常强大。在我们的实验室里，Kurtis Keller是机械工程师，他与犹他大学的Sam Drake合作，设计一种新型的头戴显示装置。我们当时在犹他大学（UU）和北卡罗来纳大学（UNC）之间运营着实时的、高带宽的视频会议系统。我们的视频会议节点离Kurtis的办公室只有150英尺，一小段路。但是我们注意到，在设计过程中，Kurtis甚至不愿花这点功夫去使用视频会议系统。他在他的办公室里工作，通过电话联系；他和Drake在他们的工作站上画图。

视频会议

视频会议曾被夸张地宣传成“改变游戏规则”的工具，现在已经广泛使用，但发展速度和广泛程度都远远不像预期的那样。为什么这么慢？在早期的日子里，低带宽导致了低帧数，给人的体验很不自然。既然现在能够提供正常的帧数，什么样的技术优势能让体验变得更好呢？

- **视野**。视频对于两个人的交谈是很好的，但如果委员会的一半成员与另一半成员开会，

就很难同时看到每一个人并真正看清楚面部表情。

- **更好地共享文档和演示。**人们希望同时看到演讲者和幻灯片或文档，而不是只能看到其中一样。人们希望在桌子上分发材料。人们希望做私人笔记和共享的标注。确实需要一个对称共享的白板。
- **更好的分辨率。**分辨率还不够好，不能让人分享完整的$8\frac{1}{2}\times11$的文本页面或看清楚面部表情。
- **更好的深度暗示。**缺少深度暗示，虽然这很少造成歧义，但不断提醒参与者他实际上不在那里。

何时视频会议最有价值？尽管当前的技术有一些缺点，但是在一些社交情况下视频会议还是比电话要好得多，虽然比不上面对面的会议。在这些情况下，面部表情和身体语言确实很重要：

- 在面试陌生的应聘者，选择最终人选时；
- 在问题对一个或多个参与者来说至关重要时；
- 当一方参与者非常不安全时；
- 当组织文化或国家文化非常不同时。

高科技视频会议。在探索最大化模拟现实的电话会议系统方面，人们已经进行了相当多的研究。我的同事Henry Fuchs已经增强了视频会议系统，提供了深度暗示，而且据说这种增强有效地促进了人们“在那里”的感觉。每个参与者的头部都得到追踪，所以产生了强大的动力学深度效果——当某人移动他的头部时，屏幕上重建的对象会根据他们离摄像头的距离而变化。而且，多个摄像头产生了3维图像，利用两个带偏振过滤的投影仪显示立体的图像。[9]

远程协作技术——被动还是主动？在远程协作的硬件和软件方面，人们已经投入了许多学术研究。这产生了许多工具和系统，其中一些投入了商业市场。这也导致了关于这一主题（和同一地点协作）的一系列会议[10]和一份值得重视的杂志。[11]

人们不得不做出结论，这些工具和系统中绝大多数都是来源于技术思想，而不是来源于对协作模式或需求的分析。实际上，在Web上快速查询telecollaboration，前面50条中的49条都是关于工具或教育，而不是关于设计中的协作。在一个图书馆的书架上，20本书中的19本是讲工具，而不是讲应用这些工具去完成任务。

这种本末倒置让我深感忧虑。这是浪费宝贵的资源（博士研究工作），而且它误导了我们有能力的学生。有用的工具创造总是从用户和任务开始的。根据我的经验，最好是在工具创造者有一个真正的用户和一个真正要完成的任务时。这样，充满缺陷的原型就不会让人满意；重要的反馈意见会马上而直率地给出。我在其他地方曾充分讨论过这个问题，这些观点至今未变。[12]

注释

1. Lohr (2009), “The crowd is wise (when it’s focused),”报告了“集体智慧”的概念，

即专门化的团队通过因特网联合起来，完成大型技术项目：

但最近的案例和新的研究表明，只是在小心设计特定的任务和刺激适合吸引最有效率的协作者时，开放创新模式才能成功。“存在这种误解，即你可以在某件事上撒上集体智慧，然后事情就会变得最好，”Thomas W. Malone（马萨诸塞技术学院集体智慧中心的主任）说，“这不是真的。集体智慧不是魔法。”

2. Clark (2006), “The Airbus saga,” 是一篇出色的新闻报道。也可以参见http://en.wikipedia.org / wiki/Airbus_380, 我在2008年9月9日访问了这一网址的内容。

3. 这项工作本身也需要一个小的架构团队。

4. Wise (1966), “IBM的50亿美元豪赌”，在项目宣布的两年之后对这个项目及其麻烦进行了非常恰当的、完整的、公平的讨论。对于协作设计，他说：“国际化的工程小组通过相当高的效率编织在一起，这让I.B.M.有理由地声称，360计算机可能是第一个完全国际化设计的产品”。

Peter Fagg是System/360项目的工程经理，他出色地完成了几十个输入/输出设备的跨部门、跨国家的开发管理工作，没有直接对这些团队发号施令。

5. Herbsleb (2000), “Distances, dependencies, and delay in a global collaboration,”和Teasley (2000), “How does radical collocation help a team succeed?”记录了分布式工作的不利之处。Hinds (2002), 《Distributed Work》, 提供了一组报告，介绍了分布式工作的各个方面的问题。

6. Ghemawat (2007), 《Redefining Global Strategy》。

7. Garner (2001), “Comparing graphic actions between remote and proximal design teams,”报告了在设计项目上，同一地点协作和远程协作的有意思的比较研究：

这篇文章介绍了一个研究项目的执行过程和发现，该项目比较了一些学生勾画设计草图的活动和结果，其中有几对学生是面对面地协作，另外几对学生是通过计算媒体工具协作……勾画草图活动（Sketch Graphic Acts）用于说明共享草图的现象和“缩略”草图的重要性——这通常是针对面对面的协作进行实验室研究，但很少有对计算机媒体协助的远程协作进行研究。

另一方面，Sonnenwald等(2003), “Evaluating a scientific collaboratory,” 不仅没有观察到区别，而且还发现科学家在每种工作模式中都找到了优点和不足：

科学协作的进化已经落后于科学的发展。协作带来的能力是否超出了它的不足？为了评估一个科学的协作系统，我们进行了一个重复测量的受控实验，比较了20对参与者（高年级的本科学生）在面对面协作和远程协作时，完成科学工作的过程和结果。

我们收集了科学结果（评级的实验报告）以调查科学工作的质量，收集了事后问卷数据以测量系统的可采用性，也进行了事后访谈以理解参与者在两种情况下对进行科学工作的看法。我们假设在远程协作的情况下，学习参与者会效率较低，报告更多困难，而且较少喜欢采用该系统。

和预期相反，量化数据表明，在效率和采用方面没有统计上的重要差别。量化的数据有助于解释这个无效的结果：参与者报告了在两种情况下工作的优点和不足，并想出了一些临时解决方案来处理远程协作中观察到的不足。虽然数据分析得到了无效的结果，但从整体上来看，这种分析让我们得出结论，开发和采用科学协作系统具有积极的潜力。

8. 一位匿名作者在Economist.com (2009)上推测“周期性的下降可能正好符合因信息技术的进步而导致的商务旅行的结构化减少”。因此这种下降可能加速视频会议技术的采用。

9. Raskar (1998), “The office of the future”; Towles (2002), “ 3D telecollaboration over Internet2”; http://www.cs.unc.edu/Research/ stc/inthenews/pdf/washingtonpost_2000_1128.pdf, 我在2008年8月28日访问了这一网址的内容。

人们也通过像Second Life这样的虚拟世界来研究远程协作。参见http://blog.irvingwb.com/blog/2008/12/serious-virtual-worlds-applications.html。

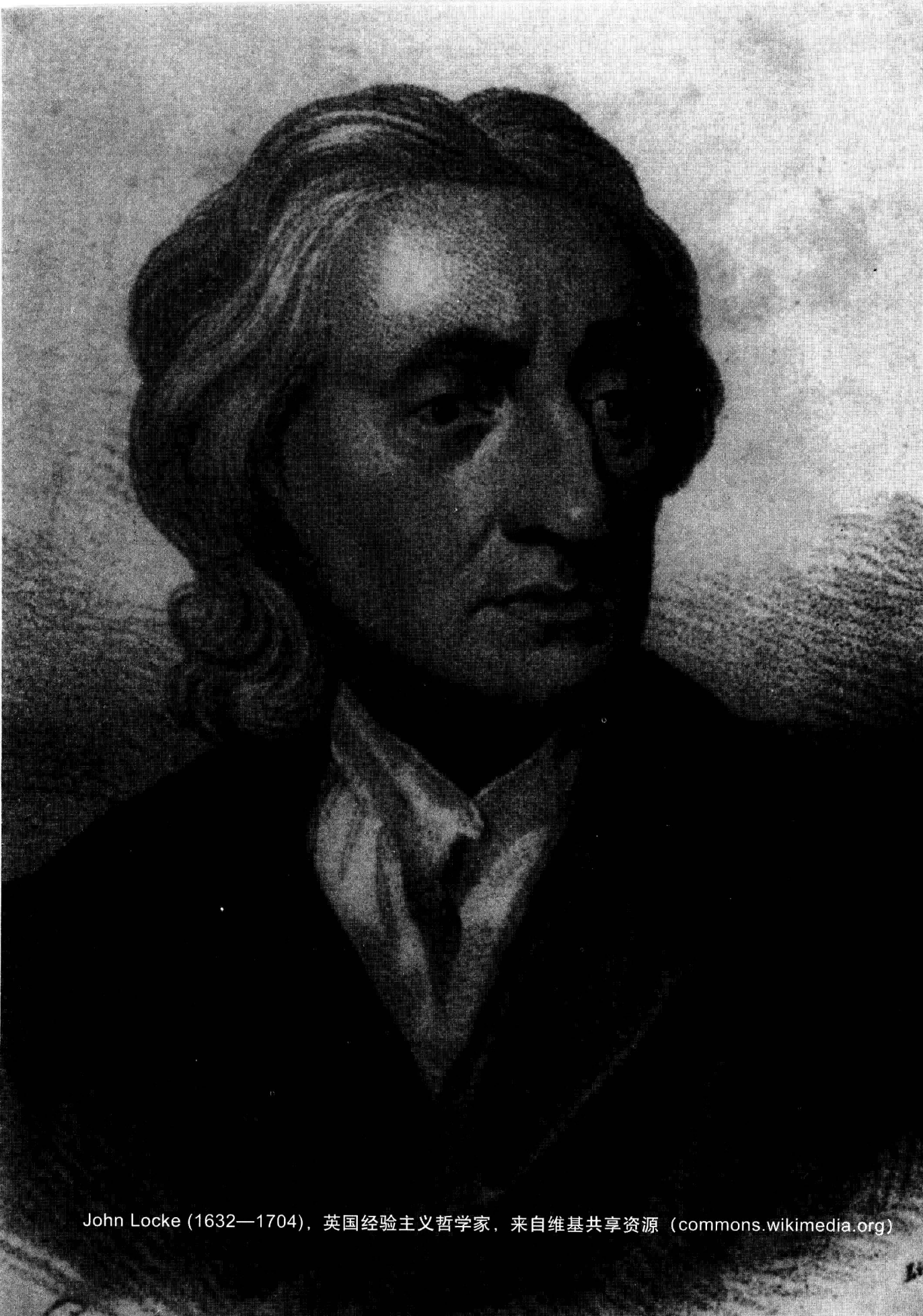
10. 参见http://www.cscw2008.org/。

11. 《Computer Supported Cooperative Work (CSCW): The Journal of Collaborative Computing》, ISSN: 0925-9724或ISSN: 1573-7551。

12. Brooks (1977), “The computer ‘scientist’ as toolsmith”; Brooks (1996), “The computer scientist as toolsmith II”。

第三部分

设计面面观



John Locke (1632—1704)，英国经验主义哲学家，来自维基共享资源（commons.wikimedia.org）

第8章

设计中的理性主义与经验主义

人人都会犯错，很多时候，大多数人都是在热情与兴趣的引诱下走向错误之渊。

——John Locke (1690)，《An Essay Concerning Human Understanding》

……理解力有两种形式：直觉与推演，无论哪一种都需要以知识作为后盾。

——笛卡尔 (1628)，《Rules for the Direction of the Mind》

理性主义与经验主义

仅仅依靠深思熟虑就能正确地设计好复杂的对象吗？这个关于设计的问题揭示出了长久以来一直存在的两种哲学体系（经验主义与理性主义）的分歧。经验主义者认为可以；而理性主义者则认为不行。[1]

这种分歧要比我们所看到的更为严重。从根本上来说，哲学问题就是人们对于人类这种创造者的本质的认识。

理性主义者认为人类天生就是健全的（也是优秀的），人们会犯错误，但可以通过教育不断完善自己。在经过正确的教育、不断成熟的经验以及足够仔细的思考后，设计者是可以做出完美无瑕的设计的。因此，设计方法学的任务就是学习如何达到完美无瑕的程度。

经验主义者认为人类天生就是有瑕疵的，会不断遭受诱惑，不断犯错。人们所做的任何事情都是有瑕疵的。因此，设计方法学的任务就是学习如何根据实验找出瑕疵，这样就可以对设计进行不断迭代了。

这种例子不胜枚举。亚里士多德（Aristotle）相信可以通过推理演绎来探求科学；因此，重的物体要比轻的下落更快。伽利略（Galileo）相信实验是必不可少的，并且过于冒失地挑战了亚里士多德的权威。

笛卡尔（1596—1650）或许最能直接阐述理性主义者观点，而John Locke（1632—1704）则清晰地提出了经验主义者观点。

时至今日，法国科学在美学逻辑结构方面胜人一筹，从傅立叶（Fourier）的热量分析，到卡诺（Carnot）的热力学理论，再到布尔巴基（Bourbaki）学派的数学大厦。与此同时，英国科学则在经验主义学派上越来越强——人们一下就能想起瓦特（Watt）、法拉第（Faraday）、海维赛德（Heaviside）、布拉格斯（Braggs）等大家。

软件设计

计算机程序是抽象的数学意义上的对象？通过证明来保证其正确性？由迪杰斯特拉（Edsger Dijkstra）领衔的理性主义者认为这个观点是正确的。[2] 人们所要做的就是深思熟虑。人们可以也应该设计出正确的软件，然后证明设计是正确的，这就足够了。[3]

但现在的程序则是纯粹的数学意义上的对象，原则上可以通过正确的思考做出完美的设计。困难并不在于设计方法而在于设计者本身。经验主义者认为人类必然会犯错误：定义对象、软件架构、对象实现（算法与数据结构）以及代码实现本身都可能产生错误。对于错误的这种坚定信念产生出了这样一种设计方法学，它包括了设计、早期原型、早期用户测试、迭代式增量实现、使用大量测试用例进行测试以及在改变后进行回归测试。

我是个铁杆的经验主义者

我之所以是个铁杆的经验主义者是因为一开始是经验使然。程序一上来就运行正确并且按照期望运行这种事在我这一辈子里只出现过两次。一次是我早期的重要程序。1953～1954年在哈佛大学研究所学习时，我们有一个学期要完成编程项目。哈佛Mark IV使用的模式是动手实践，但每组的两个学生需要花两个小时来调试、运行他们的学期项目。我的超级拍档William V. (Bill) Wright和我细致地检查了我们的1 500行的代码，检查完我都快吐了。结果程序第一次就运行正确了。

有人可能会说这个经历证明了我们是可以保证合理设计的正确性的。但我们并没有证明程序是正确的；我们只是通过模拟执行来测试。而且我怀疑是否真的有人能一直保持积极的状态，始终一丝不苟地检查我们的代码。当然了，原则上来说这是可能的。但对于真正的人和当时的软件来说，这是无法承受的。

什么经验可以保证设计的程序是正确的呢？人们已经使用过正式的证明方法来证明安全操作系统内核设计与实现的正确性。[4] 技术很适合用在这种情况下。人们需要正式证明所提供的强力保证。

当然了，即便这种保证也无法达到100%。在数学史上，很多为人所接受的证明后来都被发现是存在谬误的。[5] 正式证明并非无错的技术。它的优势在于正式证明中的推理形式不同于程序设计，这种同样的错误两次漏过审查的概率就会大大降低。

问题的关键在于我们应该使用正确性证明技术。如果内核是安全且正确的，它就能够遏制程序中的错误、漏洞以及来自其他人的恶意攻击的破坏。证明程序的正确性是一项庞大的工程，

与构建程序的工作量不相上下。没有证明能够表明程序的最初目标是正确的。[6]

哈伦·米尔斯（Harlan Mills）与他在IBM的同事开发了一种不同的设计正确性证明技术，这对我来说意义重大。在他们的“净化室（cleanroom）”技术中，米尔斯与团队将设计的方方面面都公开了，让其他热情的团队审查。在会议中，设计小组在挑战设计者的观点及其隐含的假设时倾听了设计者解释设计为何是正确的原因。[7]

正式的正确性证明通常都是不可行的，完全放弃系统验证（更加极端的情况）相当危险。依我来看，米尔斯采用逻辑论点系统但却由不太正式的小组审查是个实用的明智之举。

其他设计领域中的理性主义、经验主义与正确性

据我所知，除了软件工程外，没有哪个设计领域的设计师尝试过通过严格的形式化方法来验证正确性。或许这是因为软件像数学一样都是纯粹的思想，因此严格的证明是可行的。其他大多数设计领域最终的结果都是物理的实现，人们无法证明与原料及其缺陷和空间及其适用性相关的原理。

组织设计就像软件设计一样，因为它不涉及原料。我知道没人尝试过去证明某个假想的组织的正确性，甚至是可操作性。但《The Federalist Papers》的作者们却要通过细致入微、合乎情理的逻辑论证来证明美国宪法的可行性。虽说他们的智慧仍然在影响后代，但美国的南北战争（极为严重的系统崩溃）表明了这种阐述是不完备的。

软件工程之外的设计领域可能不会进行正确性证明，但他们却通过无数的分析与模拟技术广泛应用了设计验证。

现在的人们对机械零件进行压力、振动与声学分析。凭借实地考察与录像分析，建筑师与客户可以在设计好的建筑上模拟使用场景。对大雪与飓风进行负载压力分析测试。地震分析则提供了动态的压力测试。

计算机硬件则在电路层次、逻辑设计层次与程序执行层次上经受了大量的模拟测试。甚至连面向尚未问世的计算机的操作系统都要进行大量测试；它需要在现有主机的计算机模拟器上执行（非常慢的速度）。

大量的经验性分析所导致的必然结果就是设计过程中会出现过多的迭代。分析得越精细，就越能精确地度量出必要条件的满足程度以及约束的遵从程度。因此，针对特定目标的设计验证成为了更加直接与确定的过程。但这些分析与模拟都没有强调目标的正确性或是关于环境假设的有效性。

仅仅依靠深思熟虑就能正确地设计好复杂的对象吗？答案是否定的。实际上测试与迭代是必不可少的。但细致入微的思考还是大有裨益的。第三部分的随笔将会从某些方面介绍这种思考。

注释

1. 理性主义与感性主义是认识论的两种方法，所谓认识论就是认知的方式。在传统意义上，这两种方法的提出者相去甚远。笛卡尔拥抱经验主义科学，而Locke则认为理性主义是数学的基石。

从认识论到设计的详细阐述是我创造的，我在这里遇到了很多麻烦，可能会出现错误。

2. Dijkstra (1982)，《Selected Writings on Computing》。

3. Dijkstra (1968)，"A constructive approach to the problem of program correctness," 174-186。

4. Klein (2009a), "operating system verification,"与(2009b), "seL4:Formal verification of an OS kernel," 给出了很好的概括并取得了让人难忘的成就。作者声称这是首次从功能上完整验证了重要内核的实现。

5. 比如说，有证明表明一次矩阵乘法需要三次标量乘法。其中的缺陷在于这个假设——操作都要在向量上进行。Strassen (1969), "Gaussian elimination is not optimal."。

6. 一个知名且有启发意义的案例就是德国汉莎航空公司2904航班的事故，由于计算机控制的停止系统出现了故障导致飞机在华沙（Warsaw）机场脱离了跑道。停止系统的代码是根据规范编写的，但规范却没有正确应对意外情况。以上内容来自于http://en.wikipedia.org/wiki/Luftansa_Flight_2904，于2009年7月16日引用。

为了确保反向推力系统与阻力板只在着陆的情况下激活，部署到这些系统上的软件需要满足如下所有情况：

- 每个主要的着陆齿轮支柱必须有12吨以上的承重；
- 飞机轮子的旋转速度必须达到72节；
- 推力杠杆必须处于反推力位置。

对于华沙事故来说，前两个条件都不满足，因此大部分有效的制动系统都没有激活。第一点没有满足，因为飞机是倾斜着着陆的（为了抵消可能的风力）。这样，两个着陆齿轮达不到12吨的压力，自然也就无法激活传感器了。第二点也没有满足，因为潮湿的跑道上产生了划水现象。

7. Mills (1987), "Cleanroom software engineering."。

Wikipedia（http://en.wikipedia.org/wiki/Cleanroom_Software_Engineering，于2008年10月30日引用）很好地总结了全部方法：

Cleanroom过程的基本原则是：

基于形式化方法的软件开发

Cleanroom开发使用了Box Structure Method来指定并设计软件产品。整个团队审查过程都在验证设计正确实现了规范。

统计质量控制下的增量实现

Cleanroom开发使用了迭代方法，产品是增量开发的，不断增加实现功能。每个增量的质量都是通过预先制定的标准来验证的，验证开发过程是可接受的。不满足质量标准则不再测试当前的增量并返回到设计阶段。

统计测试

Cleanroom过程中的软件测试是以统计试验的方式进行的。根据正式文档会选择软件输入/输出轨迹中的代表性子集。接下来按统计学方式分析样本来得到软件可靠性的估算以及对这种估算的信心程度。

一个建筑团队

第9章

用户模型——错误胜过含糊

……真理来源于错误而非混乱。

——弗朗西斯·培根爵士(1620),《新工具》

明确的用户与用例模型

经验丰富的设计者经常一开始就将他们对用户的了解、使用目的以及使用模式等信息精确地记录在案。聪明的设计者还会将对用户不了解以及假设的地方明确地记录下来。

当存在多个不同的应用或是不同类型的用户时，设计者们会对每一个进行说明，然后明确地规定他们的权重以便定义用例模型。[1]

这些假设越详细、越明确就越有助于设计者们提早进行具体的思考。随着设计不断推进，后续过程就愈发需要这种思考。尽早开始可以防止错误的发生。[2]

果真如此吗

谁会在刚开始设计时就去做所有这些额外的工作呢？答案当然是没几个人会这么做了。在我看来，很多时候用例与用户模型还不够，我们还需要定义更多明确的用例与用户模型。这么做会改进设计实践。

明确的应用与用户模型的需求源自于现代化设计的特质：团队设计，复杂而非简单工具的设计。

团队设计

事实上，所有设计者在工作时脑子里面都会有意无意地想着用户与用例模型。团队设计提出了全新的要求，整个团队要拥有同样的用户模型和用例模型。这需要明确的模型与假设。

这种操练做的非常少，因为团队成员经常认为不用他人提醒，他们脑子里面想的是同样的假设。毕竟，每个人都知道企业领导在掌管与控制着团队。每个人都看过目标定义文档。人人

都是专家。

事情远非这么简单。事实上，每个人在使用类似系统时都会有不同的体验，我的体验告诉我的是典型用户的蓝图。每个人接触的都是不同类型的应用，我的经验有助于定义这个应用。如果团队没有就明确的假设起草一份常见说明，那么每个设计者都会使用他们所认为的假设，而这种假设则是见仁见智了。如果因为微观设计过小而不加以探讨，那么最后的设计就会千差万别，同时也会丧失概念完整性。

在团队设计中，不同的用例模型必然会产生不一致的设计。比如说，Operating System/360（即现在的z/OS）在某些地方使用了两种完全不一致的调试哲学——一种假设分批处理，另一种假设分时使用终端。没有人故意做出迎合这两种使用模式的决定，这仅仅反映出各个小组使用了不同的用例模型。[3]结果造成了膨胀与不一致。

复杂设计。随着工具变得日益复杂，对明确的用例模型的需求也与日俱增。即便对于铲车来说也需要明确地确定它是用于煤炭、泥土、谷物、积雪还是其他混合物；是给小孩、妇女还是男人用的；随便一个用户就能用还是得体力劳动者才行。对于卡车、电子表格以及学术建筑来说，明确的用例模型的需求力度更为明显！

此外，设计越复杂，设计者就越不可能成为领域专家来承担用户的工作。因此，隐式假定的模型就会变得越发危险了。

假如事实不可用该如何是好

当设计者开始设计显示的用例模型时，麻烦就会接踵而至：他会遇到很多不了解的东西。众多困难会迫使设计者提出他之前根本没有想到的问题。这是一件好事。

假设有人在设计一个用于安排路线与调度校车的程序产品。对两三家“具有代表性”的学校系统的实地考察会得出大量事实：时间约束、校车数量、驾驶员数量以及学生的地理分布等。但当他开始表述通用的路线与调度程序时，这些事实只会导致更多的问题。

采样系统究竟要达到何种程度？所有这些参数在整个用户集中所占据的范围是多少？他们的分布如何？

调度周期之间的变化率是多少？从现在开始的5年作为一个范围如何？10年呢？

随着问题变得日趋复杂，答案也变得越来越含糊了。如果想设计出明确的用例模型，设计者该何去何从呢？

猜测

有些问题可以通过合理的调研得到解答，但我深信一旦解决了这些问题后，设计者就应该通过猜想的频数分布来猜测或是假定完整的属性与值集了，通过这种方式来开发完整、明确且可与他人共享的用户与用例模型。

缜密的猜测胜于无言的假设。

很多收益都来源于这种“天然的”过程：

对值与频率的猜测会强迫设计者仔细思考期望的用户集。

将值与频率记录下来会引发人们的争论。批评某些具体的东西可比创造要轻松多了，这样整个团队会产生更多的输入。所有参与者都会加入到争论当中，这也会使多个设计者所设计的用户影像之间的差别浮出水面，还会揭示出其他未被发现的假设。[4]

显式地枚举值与频率有助于识别出决策所依赖的用户集属性。

更重要的是，这引发了如下的重要问题：哪个假设比较重要？重要程度如何？甚至连这种未经考虑的敏感度分析也是颇具价值的。如果事态发展为重要的决策要依靠某些特定的假设，那就非常有必要做好估算了。

最后，很多假设都是有争议且未经验证的。总架构师必须具备率领团队前行的能力。

错误胜过含糊

现在，有读者可能会问：“我怎么知道或者甚至是假设关于用例与用户的这么多细节信息呢？”答案就是“不管怎样，你都会做出这些假设的”，也就是说，每个设计决策都是以设计者对用例与用户的假设作为指导的，无论一致与否都是如此。这事实上意味着，含糊的设计者会把自己当做用户，按照自己的想法去做设计，好像他就是用户一样。但他并不是。

因此，明确的假设即便是错误的也要好于含糊的。错误的假设起码会受到质疑，而含糊的则不会。

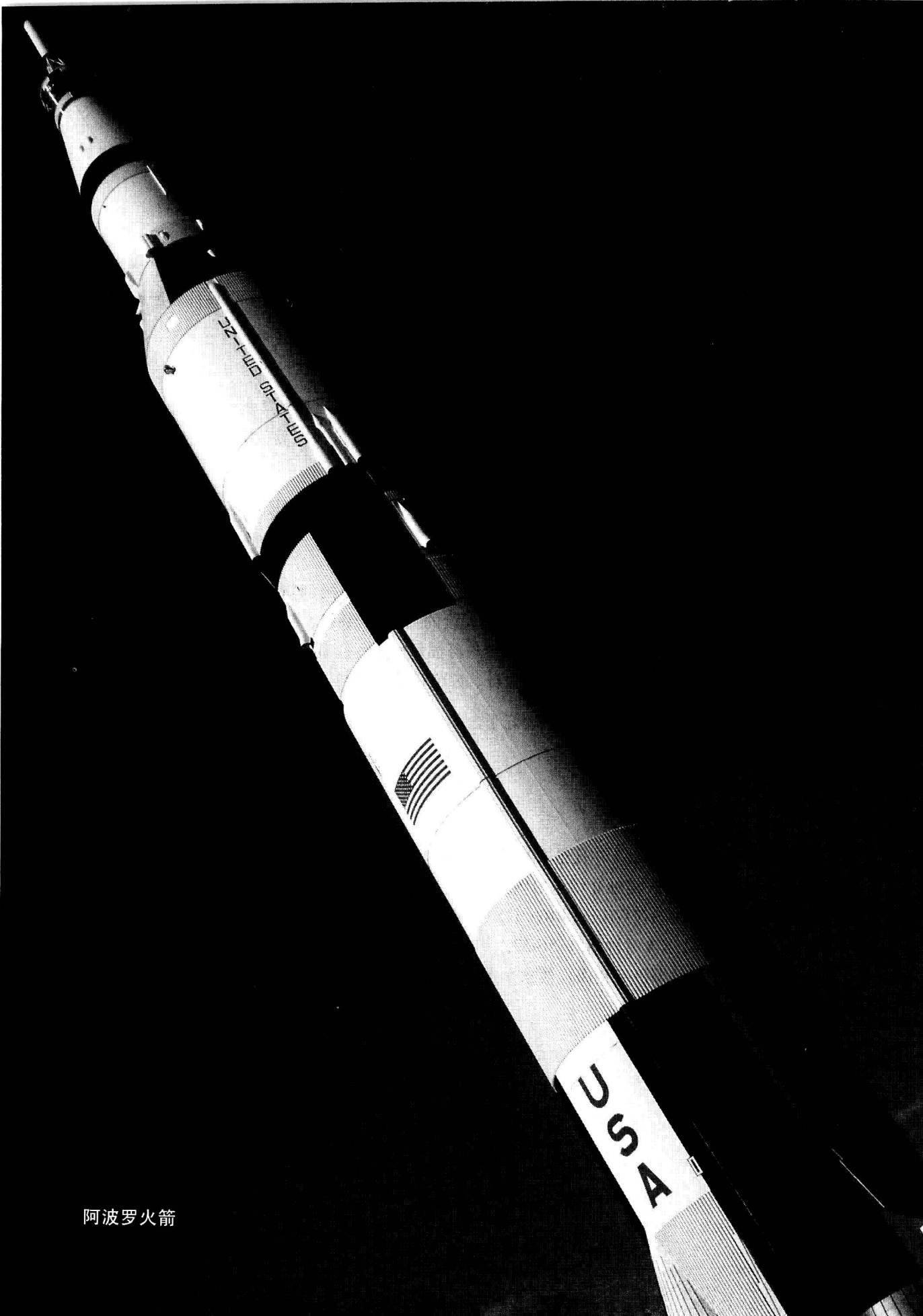
注释

1. 用例模型是用例的集合并且带有权重。Robertson与Robertson(2005)的《Requirements-Led Project Management》详细介绍了用例。

2. Cockburn (2000)的《Writing Effective Use Cases》详细介绍了用例。

3. Brooks (1995)，《人月神话》，56-57。

4. 我讲授的高级计算机架构课程要求学生将其作为学期项目。如果能够认真完成则会非常有助于设计。



阿波罗火箭

第10章

英寸、盎司、位与美元——预算资源

如果希望设计，特别是团队设计具备概念完整性，那么你应该明确地指定好稀缺资源，公开地跟踪并严格控制它。

何谓预算资源

无论何种设计都至少存在一种稀缺资源需要限量或是预算。有时需要将两种或多种资源联合起来进行优化；但大多数时候都会有一种资源是占据主导地位的，其他资源作为必要条件或是约束。经济学家把这种占据主导地位的资源叫做有限资源，而我更愿意强调设计者的这种必要行为：有意识的预算。[1]

虽然设计者经常会说代价或某种性价比就是需要优化的资源，但实际上并不是这么回事儿。因此，要想保证设计质量，特别是团队设计的概念完整性，你应该明确地指定好稀缺资源，公开地跟踪并严格控制它。

美元并非万灵丹

某些预算的关键资源并非是美元，考虑如下这些条目：

- 海滨别墅的英寸数
- 宇宙飞船或是背包中的负载盎司数
- 冯·诺依曼计算机体系中的内存带宽
- GPS系统中可以容忍的纳秒时间数
- 某小行星拦截项目中的日历天数
- OS/360设计中驻留核心的内存空间
- 会议计划的小时数
- 补助金提案或日报的页数
- 通信卫星的功率（以及存储的能量）
- 高性能芯片的发热量

- 西部农田的用水量
- 某课程学生的学习小时数
- 电影或视频的秒数，甚至是帧数
- 伦敦地铁中轨道每天使用的小时数
- 工程与维护
- 计算机架构中的格式位
- 军事攻击计划的小时或分钟数

即便美元也有不同，替代品剖析

对于设计项目来说，虽然费用是一种预算，但我们也必须考虑清楚各种各样的费用。对于大批量生产的个人电脑来说，制造费用是占据主导地位的；而对于小批量生产的超级计算机来说，占据主导地位的却是研发费用。

通常，设计者会使用美元的替代品作为预算资源。建筑师会将平方英尺作为定量资源来进行规划安排和方案设计。计算机架构师过去常常使用有限的寄存器以及各种缓存级别作为芯片面积的替代品。

替代品有很多好处：通常它们都更简单。设计者可以在了解替代品/美元比之前使用它们进行设计。它们更加稳定。虽然替代品/美元比可能不同，或是会发生变化，但使用同样的替代品可以利用人们已有的设计经验，设计者很清楚一个礼堂需要占据多少平方英尺的面积。

但替代品也会使人误入歧途，在不合时宜时人们还有可能继续使用它们。当线路长度或是针脚数量变为重要的关键资源时，芯片设计者经常考虑的还是面积因素。

预算资源是可变的

技术上的变化有时会改变关键资源。这对于粗心的人来说是个陷阱。随着芯片密度的不断增加，I/O针脚作为功能限制的角色已经被芯片面积取代了，因此它变成了定量资源。但现在功率损耗已经取代了针脚成为很多芯片设计中的定量资源。Seymour Cray曾经说过："冷却是超级计算机设计的关键要素。"[2] Gene Amdahl那时曾告诉过我，在他的设计中把片外电容当做限速资源。[3]

有些人认为众多的类已经取代了功能点成为软件复杂度与尺寸的估算目标。但经验丰富的咨询师Suzanne与James Robertson却说他们认为功能点仍然是最佳的估算目标。[4]经验丰富的开发者Eoin Woods指出：

人们想要度量两个东西：交付的价值以及需要生产多少产品才能实现这个价值。功能点很棒，因为他们会度量前者。而代码行数、类的数量等则对风格大有裨益。因此，人们可以在降低这两者数量的同时轻松提升交付的价值（来自评论家的意见）。

预算资源可以在设计中期发生变化，这是因为我们会变聪明。Operating System/360（1965年）的设计目的旨在涵盖16种计算机内存大小，从32KB（是的，是K而不是M）直到512KB以上。显然，有些内存空间必须留给应用程序，这样操作系统的驻留核心就不得不忍痛割爱了——最小要到12K。我们确定内存空间是受限的“预算资源”。但我们错了。

OS/360是可以将操作系统存储到磁盘上的首批操作系统之一。更早期的操作系统都使用磁带，甚至是卡片。拥有随机访问的驻留系统可以对功能进行扩展，这样人们就可以在有限的空间上构建系统了。“只需从磁盘上读取数据块即可”。由于访问频率陡增，系统设计团队中的每个人都认为块应该足够小，这样才能适应更小系统的1KB缓存区的要求。

我们最聪明的一个开发者Robert Ruthrauff早在OS/360项目初期就开始构建一个性能模拟器了，结果还不错，模拟器能成功运行。但最初的结果却让人大失所望！在速度第二快的计算机模型（Model 65）中，我们的程序系统一分钟内只能编译5行Fortran源代码。从那一天开始，项目的预算资源从内存字节变成了磁盘访问时间。[5]

那又如何

如果认为预算资源对于设计团队来说是一种有益的指导，那么我们应该采取什么行动呢？

明确确认

项目经理自然而然地会从列举目标与约束开始。明确确认预算资源应该是接下来要做的事情。根据之前的讨论应该知道，这通常是一种设计资源而非设计过程。比如说，技能的配备对于设计项目来说总是非常关键的要素，但它本身并非是设计的属性。

项目时间表非常重要，但它通常是项目的属性而非最终设计的属性。一方面，比如说，在截止日期非常紧张的情况下准备一份有竞争力的提案，“我们会在现有的时间内做出最棒的设计”。另一方面，如果正在进行让某个小行星转移方向的设计，那么时间表就会成为预算资源。与之类似，要想让某个全新的产品能第一个打进市场，时间表也会成为预算资源。

公开跟踪

整个团队需要清楚关键资源的当前预算。尤其是每个小组，每个团队成员都必须清楚他们设计的部分可能使用多少芯片毫瓦特、多少事务处理磁盘访问，他们能控制多少。

严格控制

无论有多少关键资源，团队领导都需要保留一些以供后续分配，就像将军会保留一些部队用于派遣到最需要的战场一样。[6]

只能有一个人来控制预算与重新预算，这是必须的。Gerry Blaauw在System/360架构的Program Status Word中就是这么来处理位的。这些再加上内存带宽与指令格式中的位就构成了架构师可以分配的预算资源。他的全局观、谨慎的节约再加上在现有架构上的创造会开发出高

效的架构。[7]

因APL语言而斩获图灵奖的Ken Iverson首先希望得到的是概念上的完整性，因此他将不同语言概念的数量作为预算资源。从而实现了团队与使用者社区针对新添加的具体概念提出大量提案的计划。他的全局观、缜密的节约再加上在现有语言上的创造最终开发出了一门优雅的语言。

现如今，Marissa Meyer（Google副总裁，决定着Google产品的发布）在设计中也运用了同样的全局观，遵循了同样的一致性以及缜密的节约来保证Google的感官。[8]

注释

1. Simon (1996)，《The Sciences of the Artificial》中也认为找到设计过程中的有限资源是非常重要的（143–144）。

2. Murray (1997)，《The Supermen》。

3. Personal communication (约1972)。

4. Personal communication (2008)。

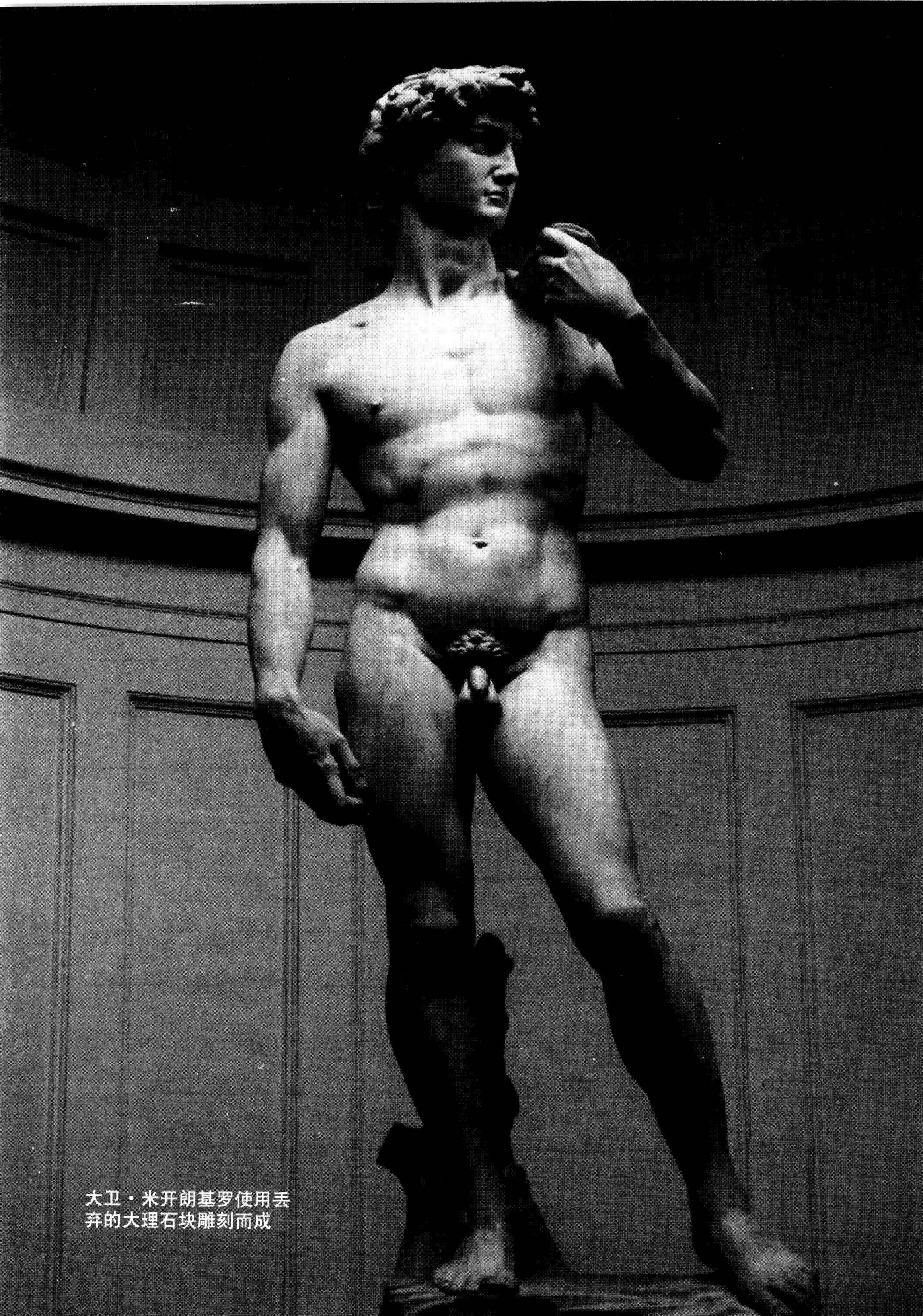
5. Digitek开发的优雅、小巧的Fortran编译器（1965年）为编辑器代码本身使用了非常密集、专门的表示方式，这样就不需要外部存储了。解码这种表示方式所损耗的时间可以通过避免使用预算约束——磁盘访问得到10倍的补偿(1969)。Brooks (1969)，《Automatic Data Processing》，第6章。

6. 一般来说，战斗是设计者所面临的最严重的情况，设计者必须快速修正设计以应对新的局面与约束。

7. Blaauw (1997), "IBM System/360," 12.4节。

8. Holson (2009), " "Putting a bolder face on Google"。

读书笔记



大卫·米开朗基罗使用丢弃的大理石块雕刻而成

第11章

约束是我们的朋友

格律是一种解放。

——来自艺术家的名言

我希望身边能有四堵围墙来保持我的生活，以免我误入歧途。

——James Taylor，《Bartender's Blues》

通用产品要比特定用途的产品更难于设计。

约束

约束可能是负担，但也有可能成为我们的朋友。约束会缩小设计者的探索空间。这样，他们就会更加专注并加快设计的过程。很多人在初中的时候不喜欢“天马行空地写东西”，我们要知道这样一个事实：去掉所有约束会使“设计”任务变得更加困难而非更加简单。

巴赫对此深有体会。Wolff曾说过：“虽然一开始巴赫倾向于穿越传统的边界，但他更愿意在既定的框架下工作并接受由此带来的挑战。”[1]

约束不仅会缩小探索空间，而且它们还会向设计者发起挑战，这常常有助于激发新的创造。现在来说说大卫·米开朗基罗。据说，因为出现裂缝，Antonio Rossellino早在25年前就将这个没用的大理石块丢弃了。结果造就了大卫不同于先前和当代艺术的新概念。有人对此感到异常气愤，想要看看米开朗基罗到底是如何处理大理石块的瑕疵的，同时还想了解这又是如何激发出他那独特的艺术概念的。

Christopher Wren修筑的伦敦教堂为我们展现了另一个鲜活的例子。Wren受命重新修建因1666年的大火而损坏的50个英国教堂，他感到这种约束难以忍受。每一个教堂的地点都是确定的，环境是确定的，还有一些地基尚存，这都是他所面临的约束。此外，英国教堂的圣坛必须要面向东方。现如今，人们可以参观第二次世界大战之后所余下的27个英国教堂。看看它们的位置与问题，想想方向的约束，思考Wren当时是如何解决这些问题的。

在北卡罗来纳山脉上，蓝岭景观道路高架桥的设计者们必须尽可能少地触及地面以减少环

境破坏，而结果却是相当优雅。[2]

不完全如此

设计任务中的人为约束可以很容易地缓和下来。在理想情况下，他们会将设计者推向设计空间中未曾预料到的角落，进而激发出创造力。但任何约束集都有可能将设计者推向空角落，这时是不存在可行性设计的。

因此，人们必须细心地区分：

- 真正的约束
- 已经过去的、曾经真正的约束
- 误认为真正的约束
- 有意制造的约束

过去的约束。有经验的设计者会像狮子一样习惯于度量笼子的范围，他会发现自己遵从习惯的约束，但随着技术的不断发展，这些习惯可能已经不适用了。第9章提到的《人月神话》(1975）中曾经说过“在容纳5磅重量的麻袋里放10磅东西”，现在看起来这就像笑话一样。这个例子说明了如何将软件压缩到狭小的内存空间中。这在1965年是非常重要的，但在1975年时就差很多了，但很多程序员依然会对较小内存心有余悸（当然了，内存大小对于嵌入式电脑来说非常重要，尤其是“手机”所用的众多强大系统更是如此)。

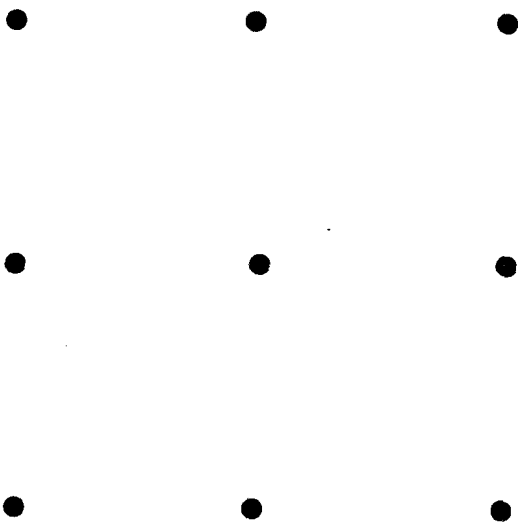


图11-1　让人误解的约束谜题：使用一根实线穿过所有9个点，实线最多可分为4段

误解的约束。这些约束更加不明显。图11-1是个很典型的示例（参看图11-5以获得答案[3])。第3章介绍的边界线约束是另一种情况。

要想在7次而非8次乘法中增加两个2 x 2的矩阵，你必须跳出使用矢量操作这个误解的约束。

在为联邦航空管理局（FAA）设计IBM 9020计算机系统时就经历了一场痛苦的折磨。MITRE公司的系统架构师作为FAA的代理人着眼于超可靠的系统。他们指定了配置，如图11-2所示。

到目前为止一切都还好。IBM团队在投标这个需求时发现新的System/360半集成电路技术非常适合于这种单元可靠性需求。

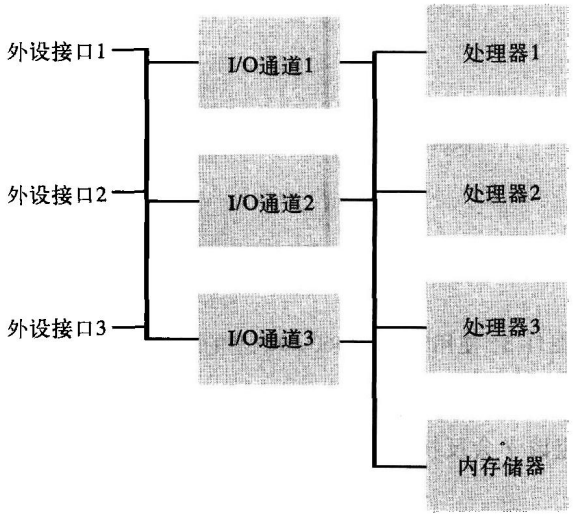


图11-2 为1965年FAA系统所指定的带三重模块化冗余结构的处理器与I/O配置示意图

S/360 Model 50是一款中等级别的计算机，它超额满足了处理器对速度与可靠性的要求。Model 50的I/O系统与处理器的内存及数据通路的实现方式是一样的，但却使用了不同的微码设计，它漂亮地满足了I/O控制器的需求。

因此，IBM的工程师们按照图11-3的方式设计了系统。

经过仔细分析会发现图11-3的配置已经超额满足了系统性能与所有的可靠性需求。但它却被拒绝了。

上图并没有显示指定的配置拓扑。MITRE系统架构师错误地坚持特定的拓扑是必要的约束——虽然他们实际需要的是功能与可靠性而非拓扑。因此，IBM按照图11-4的方式进行了投标、构建并交付了配置。直接的可靠性分析表明这种配置实际上是不如图11-3的配置可靠，因为有两倍以上的组件及更多的连接器可能会发生故障。但它却满足了指定的约束！

对于纳税人来说，系统的价格是不可思议的。除了花费以外，政府为图11-4中的系统支付的费用与图11-3中的是一样的。IBM实在太想签下这个合同了！当然了，使用期内用电、制冷以及维护的费用是不同的。

在开始设计时，你需要确定到底需要哪些东西而非如何得到它们。

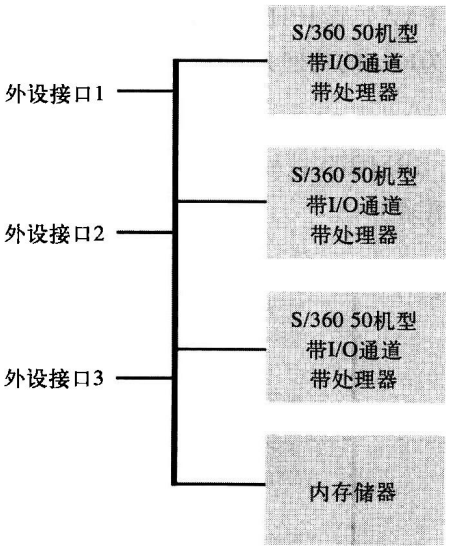


图11-3　使用System/360 50机型的FAA系统初步提案

如果实现方法成为约束，那更好的解决方案就是终止这一切。出于产品与用户的考虑，设计者在遇到虚假约束时应该学会反抗！

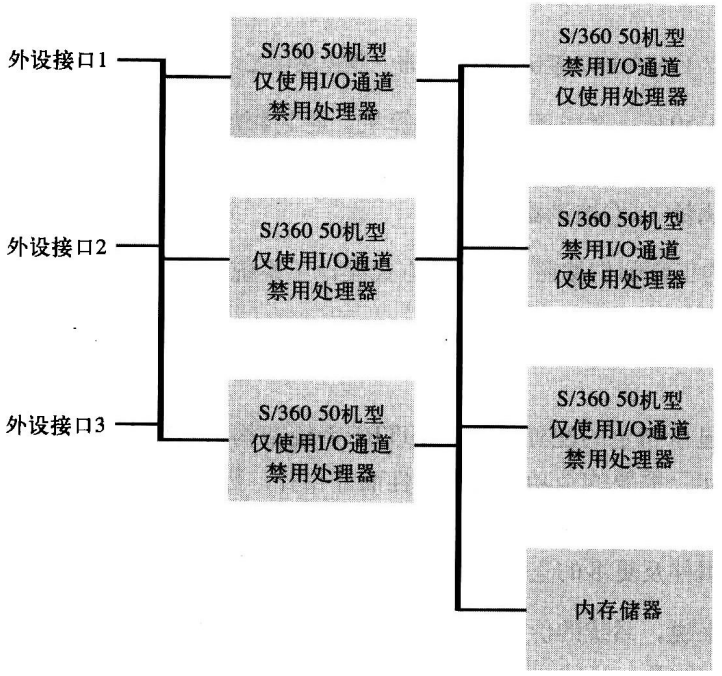


图11-4　实际交付的IBM 9020 FAA系统

设计悖论：通用的产品要比特定用途的产品更难以设计

我之前就曾说过设计中最难的地方在于明确确定到底要设计什么。上面曾说过约束是我们的朋友，它们会使得设计任务变得简单而非困难。一般来说，目的越具体，设计任务就会

越容易。

乍一看这种观点是错误的。人们想当然地会认为“设计一个1 000平方英尺的房子”要比“为一家拥有一男一女两个小孩、位于北卡罗来纳教堂山市且朝北的家庭设计一个1 000平方英尺的房子”容易一些。

当然了，从某种意义上来说，前者确实容易一些——人们基本不会批评这个设计。如果没有约束，就不存在优秀的准绳。通常对于一件普通的事情来说，通用的设计要比特定用途的设计简单一些。

然而从整体上来说，如果想要得到优秀的设计，那么后者更容易达成目标。任何设计过程都是以设计者对目标与约束的详细阐述和具体化开始的。首先要做的事情是缩小设计范围。指定目标的约束越多，完成任务的可能性也就越大。

匆忙完成的通用设计。怎么会这样呢？考虑一下计算机架构的例子。通用计算机很容易为人所理解。现在已经构建并销售出了100多种样本架构。每个人都知道它们的作用是什么。好的设计者能够在几天之内模仿出一个。架构决策集是非常明确的：

- 指令格式
- 地址与内存管理
- 数据类型及其表示
- 操作集
- 指令序列
- 监控工具
- 输入输出

设计特定用途的计算机架构。设计特定用途的计算机显然会多出很多额外的工作。设计者必须学习应用。它为什么这么特别呢？相对的操作频率是多少呢？对于客户来说，性能、费用、可靠性、重量等必要条件分别占多少权重呢？事实上，设计者不得不明确应用的特征。

设计优秀的通用架构。设计者还需要明确用例模型才能为通用架构进行恰当的设计，但用例模型却难以绘制出来。事实上，设计者必须理解全部应用，确定每一种应用的特殊需求并掌握其平衡性。科学计算机强调矩阵代数与偏微分方程式；工程计算机强调数据简化与公式求值；数据库查询则强调最优的磁盘使用率。设计者必须理解每一种应用。

接下来需要指定它们的权重：

- 在所有应用集上
- 在所有预期使用的机器实现上
- 在新架构必须经历的数十年生命周期中[4]

类似地，随着设计不断进行，必须将得到的结果与每类用户假设的特征进行比较。当设计完成并开发好原型后，必须让每类用户对原型进行测试。

我总是让参加我所开设的高级计算机架构课程的学生去做特定用途的架构项目。他们不可以使用陈词滥调来敷衍我，应用与用户分析必须精确。通常，他们完成的项目都很棒，然而却无法在给定的时间内设计出优秀、无约束的通用架构。这个任务要比深度分析容易得多，因为后者要求实现严格的通用设计。

软件设计。同样的悖论也存在于软件当中。相比于人们在通用编程语言中费力寻找的表现力、通用性以及简约性之间的微妙平衡，特定用途的编程语言就显得非常直接了。约束很容易在特定用途的设计中得到利用。

空间设计。这个悖论也存在于建筑空间中。设计华丽的卧室要比公共居住空间容易得多，这恰恰是因为公共空间具有更多功能，因此要了解更多的使用场景，另外，设备的选择余地实在是太多了。[5]

类似地，设计专业实验室要比计算机科学楼的公共大厅容易得多。

小结

既然约束是设计者的好朋友，那如果任务最初看起来没有约束，首先就需要仔细思考想要的用户与用例模型到底是什么，这样就会发现一些约束了，这对于设计者与用户来说都是好事。

注释

1. Wolff (2000)，《Johann Sebastian Bach》，387。当委员会或现有的表演才能不能构成足够的约束时，巴赫有时会使用人为约束来激发创造力。重复使用的BACH主题就是一个例子（重复演奏降B、A、C和B本位音）。我并不建议在工程或软件领域中采取人为约束的手段来激发创造力。

2. http://www.blueridgeparkway.org/linncove.htm，于2009年7月18日访问。

3.

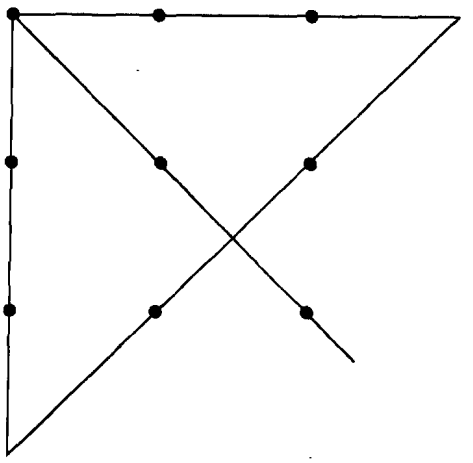


图11-5　9点谜题的答案

4. 根据几代实现的情况来看，我们一开始预计System/360架构只能存活25年（Brooks (1965)，“The future of computer architecture”）。用来保护编程语言、计算机架构与操作系统架构的力度超出了我们的想象。45年后，IBM的z/90仍然嵌入了System/360架构，该架构寿终正寝的日子还很长。今天使用的Fortran（1956年）就很好地证明了那种持续不断的力度。

5. Robert Adam（1728—1792）为伦敦附近的汉普斯蒂德Kenwood House详细设计了家具的每一个细节，甚至连门把手都设计好了（来自《Kenwood旅行指南》）。

蒙蒂赛洛：杰弗逊（Jefferson）采用了帕拉迪奥（Palladio）所采用的罗马风格

第12章

技术设计中的美学与风格

坚固、效用与情趣

——Marcus Vitruvius (公元前22年),《De Architectura》

风格是思想的外衣，良好的思想就像是穿着考究的绅士一样更具优势。

——Chesterfield勋爵 (1774)，私函第240封

技术设计中的美学

Vitruvius曾经有句著名的断言：好的建筑架构要满足“坚固、效用与情趣”的要求。[1] 虽然从纯粹的功能主义观点来看，坚固加效用就足够了，但情趣的重要性丝毫不逊色于这两者。

全人类的经历都证实了他的断言。我们总是希望公共与私人建筑能够满足人们心里对美的要求，而且我们也愿意为了这个目标付出更多的努力。

穴居人会在洞穴的墙上绘制图案，美国原著民会装饰他们居住的帐篷。史前英国人使用手工雕刻品装饰瓷器。几个世纪以来，模型、雕刻、瓷器、镶板以及彩绘都会对坚固的结构与居所的便利性起到补充作用。美的标准差别是非常大的，从华丽到倾斜，虽然受到文化与时代的影响，但是视觉上的美总是建筑设计的目标。

美与艺术在技术设计中分别担当何种角色呢？汽车、飞机与轮船都有确定的外形，因此能展现出视觉上的美。但这并非全部。我们发现自己总在谈论“优雅”与“丑陋”的编程语言。当提到“整洁”的计算机时，我们指的不仅是工业设计上的可视化效果，还包括逻辑结构的某些属性。

我们在看到或使用某些设计时会感到很愉快，而其他具有相同功能与稳固性的设计则不会给我们带来这种感觉，我们对这一点是很在意的。情趣可能是可见的、可听的、可闻的或是可触的。它甚至有可能是纯粹的智力，就像是漂亮的证明或是良好的思想。我们的语言会以多种

方式捕获到这些——当我们不经意间提到“优雅”的程序时，听众会知道我们的意思。[2]

何谓逻辑美

简约

“优雅”需要简约。在数学中，对优雅的一个定义就是“使用更少的元素完成更多的处理”。这当然也适用于证明。很多数学教科书作者认为它也适用于阐述说明。光是清晰、充分还不足以称为好的阐述，需要明确需求并使用示例加以说明。

人们倾向于将简约作为编程语言的设计原则。计算机设计必须将简约放在重要的位置上，即使用更少的元素才行。[3]

比如说，Lisp就有一个很精巧的内核，但对可扩展性与可组合性的优雅支持使其具备了强大的能量。相反的是，Visual Basic就更加复杂一些，但却难以扩展。

这还不是全部。光有简约还不够。为计算机添加一个“多余”的元件，比如变址寄存器，能从根本上改进性能与性价比。Common Lisp对基本语言进行了扩展，使得程序员能够轻松上手开发。

van der Poel只使用一种操作码就设计出了一种计算机。[4]每条指令都执行同样的操作。他证明了这种操作的充分性——他的计算机可以完成其他计算机所能完成的一切。但却难以编程。颇具讽刺意味的是，使用它所带来的情趣类似于做出纵横字谜游戏一样——人为增加的复杂性，并不实用。

实际上，要想高效使用van der Poel机器需要掌握一整套特殊的语言。van der Poel编写了一些子程序与宏操作，这样其他人就能使用更高层次的语言编程了。

类似的情况也发生在APL上，这是一种优雅、强大的编程语言。APL中存在多种编程风格——从清晰与直接到复杂与过度紧凑。虽然操作符都是非常直接的动作，但有时需要使用特殊的语言组合来实现高频率的功能。读者可以从《Computer Architecture》的A.6节，程序A-41中找到相关示例，该示例涉及“求偶数”、平分数字、向下舍入以及平方计算（Blaauw和Brooks(1997)）。

有时，人们想要看看一行APL代码到底能承载多少功能，虽然这很复杂，但却符合语言习惯。“一行”代码实现的功能会让人获得满足感与自豪感。但我想重申的是，这仅仅是纵横字谜游戏而已，绝非优雅的设计。程序语言存在的目的是为了简化程序员编写与阅读代码的工作量，绝不能成为谜题一样的东西。[5]

结构清晰

光有简约还不够。人们还要求语言或计算机架构具备某种程度的坦率性。从想要说到如何说之间应该有个直达线路才行。

人类的自然语言可以看做是用来满足实际需求的一种融合，但它却并不简约。Shannon (1949)认为英语的冗余度高达50%。[6] 与这种冗余不相上下的是人类语言还有习语，就像编程语言一样。

结构。技术设计中的“优雅”要求设计的基本结构性概念要易于为人所理解，否则逻辑就应该是直接且易于解释的。

隐喻。“优雅”与可理解性都是通过使用熟悉且简单的隐喻实现的，尤其在设计对象的用户界面上更是如此。Macintosh操作系统的“桌面”就是最好的例子。嵌入到Excel与Lotus中的VisiCalc所发明的“电子表格”也是一个好榜样。

一致性

Gerry Blaauw与我曾研究过这个问题：“什么才是好的计算机架构？”或许看看小巧的处理方式就能解释更大的问题：“什么才是好的技术设计？”我将在下一节根据《Computer Architecture》（1997）的1.4节的内容来总结。[7]

什么才是好的计算机架构

没有包含所需功能的架构是错误的架构。但即便包含了所需的功能，架构也有可能不太合适。或者是整个架构过于复杂以致难以学习并记住功能与规则。易于直接使用的架构通常叫做整洁的架构。

Blaauw与我相信一致性应该是所有质量原则的根基。好的架构应该是直接的，人们掌握了部分系统后就可以推测出其他部分。[8]

比如说，在一个指令集中只包含一个平方根操作符几乎就能将操作定义完整。数据与指令格式应该与其他浮点数运算操作符相同。精度、范围、凑整，它们的意义应该与其他结果的处理方式相一致。甚至是取负数平方根也应该与被零除采取相同的异常处理方式。

但真正一致的解决方案是很难识别出来的。对于计算机架构来说，有一些简洁的标准、简单的代码生成以及适用性可用于众多的实现当中。

导出原则。从一致性角度来看，有3个主要的设计原则：正交性、适宜性以及普遍性。

正交性：不要将独立的东西链接起来。正交函数的一个变化对集合中的其他任何函数不应该产生显著影响。比如对闹钟来说，功能集中应该包含发光表面与警报器。如果只有表面亮起才操作警报器那就违背了正交性的要求。在我们的书中给出了计算机架构中违背正交性的大量实例。

适宜性：不要引入无形的东西。满足必需的功能是适合的功能。对于汽车来说，方向盘、速度控制器、车灯以及挡风玻璃洗涤器都是适合其目的的。

适宜性的反面就是外部组件。汽车变速杆就是一个很好的例子。滑动的齿轮并不适合于驾驶；用户界面的外部组件都来自于汽车的这种实现。

计算机中适宜性的一个例子就是零在两种补码符号中的唯一表示。与此相反，有符号数与补码会在零前添加一个符号，这种差别来自外部。我们需要使用更多规则来明确在数学以及与零相关的操作中有符号零的行为。这通常会在使用时导致无法预料的行为。

简约是适宜的一个子集。另外一个就是透明性，该属性可以保证功能实现时不会产生可见的副作用。在计算机中，使用流水线的数据通路实现应该对程序员透明。

普遍性：不要限制固有的东西。普遍性就是通过某种功能实现众多结果的能力。它能够反映出设计者的专业素养，用户可以超越设计者的想象发明出新的东西，这种需求已经超出了设计者的能力。设计者应该避免将某个功能限制在他自己的使用观念中。如果你不知道，那就让它自由而去吧。

Intel 8080A有个名为重启的操作。实际上，它用于中断后重启。但设计时考虑到了足够的普遍性，现在它最常用于从子程序中返回。

实现普遍性的最常见方式是开放的结尾（留下未来开发的空间）、完整的功能集、分解功能为正交组件以及使用组合。

一致性的更多优点

一致性是通过不断加强与自我学习实现的，因为它会证实并鼓励我们的期望。它还会解决易用性与易学性之间的冲突。易学性需要简单的架构，正如定点运算一样。易用性需要复杂的架构，正如浮点运算一样。当设计者将定点运算作为浮点运算的一个子集时，用户对架构的理解就会自然而然地得到提升。

技术设计中的风格

从一段古典乐的中间开始收听时，有经验的收听者通常会猜测乐曲的年代和作曲家，即便他从来没有听过这个乐曲。当看到一幅不熟悉的绘画时，我们经常会说“看起来好像是（伦布兰特）Rembrandt”或是“荷兰的黄金时代”。二战期间的英国WRENs掌握了如何识别Axis广播公司典型的摩斯电报码“fists”，因此当军队行军时能够获悉他们所属的部队。桥梁、汽车、飞机以及计算机架构也都一样。据说Seymour Cray设计出了一台特殊用途的计算机。

情趣的一个驱动力就是风格（后面有定义）。两种风格的组件影响到了情趣：影响情趣的一致性与风格本身的内在质量。在架构、音乐、烹饪以及计算机领域中，没有一种风格会吸引所有人，混杂的风格让人叫苦不迭。[9]

我之前曾说过概念上的完整性是设计最重要的属性。当然了，设计最重要的完整性就在于整体结构，这是设计的骨架。但细节中也需要遵循一致的风格，就像皮肤一样。

一致的风格就是组件，即便产品的概念完整性只是个“裙子”而已。它超越了情趣——它有助于设计的可理解性。反过来，这又会简化学习、简化使用，在误用之后还能有助于回忆，简化维护、简化扩展。

风格在各类设计、流派中都占据举足轻重的地位。

何谓风格

确切来说，能够让设计者认可产品的典型方式到底是什么呢？这个问题实际上很难回答。

定义。就我们所考虑的情况来说，《牛津英语词典》对风格的定义如下：

14. 文学作品中属于形式或是表达的那些特性，而非思考或表达的物质特性。

21. 有经验的建筑、执行或是产品的特定模式或形式；艺术品的创建形式，人们的时间、地点或是艺术品的特征。

《韦氏英汉字典》(1913版)：

4. 特别是音乐或任何美术的展现形式，某种想法或实现某个结果的特征或特别的模式。

Akin [1988], "Expertise of the architect":

作为设计者个人与专业性选择的风格是一种媒介物，有助于限制设计问题中可能具有的过多自由度。

一种细节特性。我们注意到同样的艺术家的不同作品所绘制的主题是不同的，音乐作曲的流派与主题也是不同的，但在风格上却是类似的。与之类似，Frank Lloyd Wright所设计的橡树公园教堂与其他建筑师所设计的教堂在某些部分与布局上则有相似之处，但它与Wright的住所在线条、细节、装饰物以及颜料上则有着紧密的联系。不管风格是什么，它与设计细节关系密切而非主要目的或是强力。[10]

假设：将脑力劳动最小化。所有设计、创造都会涉及大量的微观决策。人类的习惯倾向于减少脑力劳动，以此来降低每天做决定的负担。如果这是人类与生俱来的特质，那么它当然会继续存在于我们的创造性活动中。如果没有足够的原因支持我们这么去做，那就会每天都做同样的微观决策。一直以来，微观决策表示着我们的工作并具备自己的特质、特殊性，这会使我们具备可分辨的能力。

微观决策的一致性。人们希望微观决策不仅在时间上是一致的，类似的决定也应该是一致的。在相关的微观决策中会有同样的因素，同样的思维自然而然地会以一致的方式衡量它们。

A Frank Lloyd Wright倾向于在装饰与结构中使用线性元素来代替弯曲的。A Seymour Cray对他早期的计算机选择了最高的性能而非兼容性。

清晰的风格。设计者在大范围的宏观与微观决策中获得了一定程度的一致性，我们说它有清晰的风格，这意味着能以简约的方式描述它。它使得识别变得更加简单。

甚至连复杂的结构也能展现出清晰的风格。Wright的架构既简约又清晰，但它们却并非具有相同的属性。

如果涉及决策中没有达成多少一致性，那么我们就称这种风格为不透明或是混乱的。在某种程度上，一致性会带来清晰性，清晰性会带来情趣。[11]

我的工作定义：

风格是一套不同、重复的微观决策，即便上下文可能不同，但每一个决策的制定都是采取了相同的方式。

此外，相关的微观决策需要使用相关联的方式来制定。

风格是确确实实存在的。一些酒吧告诉我们某个作品是巴赫的或莫扎特的或舒伯特的。某个知名展会同时带来了很多伦勃朗（Rembrandts）以及当时被认为是伦勃朗，后来发现是赝品的很多作品。与会专家能分辨出真品，甚至连外行都能说几句。[12]

许多知识渊博的人很容易就能从Gerry Blaauw或是Gordon Bell中找出Seymour Cray计算机。[13] 未签名的《Federalist Papers》的作者最后是通过他们散文的细节信息被识别出来的。[14] 程序员能够分辨出彼此的代码。[15, 16]

风格的属性

不论设计媒介是什么，风格都有一些相通的属性。

制定规范的代价是高昂的。首先，明确指定风格需要大量的规范。《Chicago Manual of (英文散文) Style》使用了984页。Fowler的《Modern English Usage》(1926) 使用了2 800个词语来定义the这个单词。我们对巴赫风格的明确定义也将会使用大量的信息。

规范是层次结构的。其次，任何风格规范（无论是明确还是含蓄）本质上都是层次结构的。考虑一下英文散文：

- 方言与措辞
- 人物、时态、礼节、生动的颜色、热情的语调
- 平衡与节奏——韵律
- 使用——比如说性别代名词
- 标点符号
- 组合布局——字体、空格等

本书的组合风格（本书的附赠网站上有[17]）涉及书的样式、章节风格、标题、段落、字体以及其他元素。

风格进化。最后，风格一直都在进化，甚至连个人风格也是如此。专家能从Turner的早期作品中挑出晚期作品。[18] 大量的风格，比如哥特式建筑，进化非常明显，我们都可以分辨出早期的、装饰过的以及垂直的。此外，风格中的时尚元素进化得更快，无论是流行音乐、青年话语还是17世纪的英国公园莫不如此。

要想获得一致的风格——记录下来

设计风格是通过一套微观决议制定的。清晰的风格反映出了一致的集合。清晰的风格未必是好的风格；但混乱的风格则绝非好的风格。[19]

有抱负的设计者必须要为风格的一致性而奋斗。设计团队需要更加努力地工作。只编写10页内容的独立作者会保持清晰的风格。同样的作者如果写一本书则需要在进行过程中记录下来某些风格上的微观决议以保持一致性。此外，如果是几个作者一起合作，那么甚至连10页纸的内容也需要风格表格（或是细心的编辑）。

在大多数设计上下文中，很多风格上的决策早在一开始的时候就确定好了。技术文章具备日报式的风格；书籍具备出版物式的风格。汽车设计者会假设一个巨大的SAE标准集外加偏爱使用的制造弹簧、螺母以及螺钉的公司名录。操作系统设计者会有一个标准的子程序库。

然而，每个特定的设计都会引起大量自由的微观决策。Gerry Blaauw与我发现我们合著的《Computer Architecture》已经发展为19页的Writing Style Sheet。[20]这当然会对Chicago Manual of Style与Addison-Wesley自己的风格文档起到补充作用。虽然很广泛，但它们留下了太多无法回答的问题，我们只能自己来解决了。比如说，我们该如何参考特定的计算机？厂商和型号呢？所有这些都是首次遇到的吗？或者说在每一时期都是首次出现的吗？

当然了，设计团队必须将设计恰当地记录下来，无论是工程图纸、构建蓝图还是用户手册都是如此。他们还必须说出为何要获取设计者的意图，这样后续的维护者才不会遗漏掉重要的内容。这才是最终的产品。团队还必须在维护阶段保持概念上的完整性，将管理与构成可视化设计的各种微观决策记录下来。

如何获得良好的风格

对策相当简单；方法要直接；工作要努力。

有目的地去学习其他设计者的风格。练习另一种风格。这会强迫你对细节进行缜密的思考并明确想法。它还会得到重要的成果——思考Respighi的《Ancient Airs and Dances》或是Fritz Kreisler的《Classical Manuscripts》以及他的《Praeludium and Allegro (in the Style of Pugnani)》[21]。

做出有意识的判断。为你所喜欢的风格写出评价与原因，喜欢它们的哪一方面以及为什么。

练习，练习，还是练习。

修正。看看不一致的风格。

仔细选择设计者。为你的产品选择拥有清晰风格与高品位的设计者，根据他们之前的工作做出判断。

注释

1. Vitruvius (公元前22年)，《De Architectura》。

2. Gelernter (1988)，《Machine Beauty》，对美的感觉给出了强大的论据，不仅仅是分析，对设计也具有指导作用。

3. Steve Wozniak (2006)，《iWoz》，自豪地宣称相对于其他机器来说，Apple I只使用了2/3的组件。

4. van der Poel (1959)，"zEBRA, a simple binary computer."。可以通过Blaauw与Brooks (1997)的《Computer Architecture》13.1节找到说明。另请参见van der Poel (1962)，《The Logical Principles of Some Simple Computers》。

5. 我们在《Computer Architecture》程序9-8（第511页）中给出了不同的APL单行代码(Blaauw与Brooks (1997))。

6. Shannon (1949)，《The Mathematical Theory of Communication》。

7. Blaauw与Brooks (1997)，《Computer Architecture》。

8. Blaauw (1965)，"Door de vingers zien"；Blaauw (1970)，"Hardware requirements for the Fourth Generation."

9. C. S. Lewis (1961)，《An Experiment in Criticism》强烈认为风格的优秀与层次（"知识分子"与"低俗之人"）是互不相干的。

10. Alexander (1977)，《A Pattern Language》是个重要的架构方面的示例。

11. 一些人工与模仿的让我满意的清晰风格的例子：

- 位于巴塞罗那的露天西班牙建筑博物馆，西班牙村
- 迪斯尼世界的未来公园与国家展览馆
- 杜克大学的哥特式校园
- Jefferson位于美国弗吉尼亚大学草坪上的10个展览馆，每个都使用了独特的设计
- Sidney Smith的18世纪散文
- 引用的Respighi与Kreisler的作品

12. 一个作品来的时候被当做模仿品，最后被证实是真的并走进了芝加哥艺术馆。

13. 比如说，查看Blaauw与Brooks (1997)的《Computer Architecture》中Computer zoo节的14章（Cray）与15章（Bell）以及12.4节（Blaauw风格）。

14. Mosteller (1964)，《Inference and Disputed Authorship》。

15. Lewis (1947)，《Miracles》，第15章。

16. Chen (1997)，"Form language and style description"认真分析了何谓风格？并尝试度量它。

17. 参见网址：http://www.cs.unc.end/~brooks/DesignofDesign。

18. 大多数艺术家的风格都在不断演化。北卡罗来纳艺术博物馆展出的"Monet in Normandy"（2006—2007）强调了过去几年他的风格的惊人演进。

19. 有人喜欢Gaudi设计得非常棒的巴塞罗那圣教堂，有人却对其嗤之以鼻，但毋庸置疑的是其风格在概念上的完整性。

20. 发表于本书的网页：http://www.cs.unc.end/~brooks/DesignofDesign。

21. Kreisler首次发表于1905年的《Classical Manuscripts》，说他发现它们位于法国南部的古老修道院里面。他对自己的名字经常以作曲者的身份出现在音乐会中感到很尴尬。1935年，他承认是自己作的曲，再版时就成为“in the style of.”了。

来自于《Palestrina Mass》的巴赫手抄本
柏林国家图书馆

第13章

设计中的范本

……当你寻找某些东西时，只有事先了解了它们才有可能找到。如果不了解，你都不知道去哪里找，或者是不知道是否已经找到了想要的东西。这再一次证明了当寻找某些东西时，架构师的脑海中还是想着过去的方案与风格概念……

——Bill Hillier与Alan Penn (1995), "存在一种领域无关的设计理论吗？"

很少会有全新的设计

这当然有些玩笑的意味了！人们很少会进行全新的设计。想想首个地球轨道卫星、首个移动电话、首个WIMP界面、首个航空集散站以及首个超级计算机的设计吧！

共通的东西占多数。但在通常情况下，即便是创新的设计也是来自于之前为完成类似目的所做的成果，并且构建在类似的技术之上。设计者自己之前可能设计过类似的东西；如果没有，他一定学习或使用过。

那么范本、判例在设计中的正确角色是什么呢？设计者应该如何学习并使用它们呢？每个设计领域都应该积累一些可供大家使用的范本吗？如何做？谁来做呢？

范例的角色

范本为新设计提供了安全的模型以及设计任务的隐式检查列表、潜在错误的警告，还可以作为全新设计的出发点。

因此，好的设计者应该投入大量精力来学习判例。Palladio不仅研究了维特鲁威人（公元前22年），他还远赴罗马考察并记载存活下来的遗迹，学习古罗马经过世代演进而留下的最成功的概念与比例。经过这种单调、无名的辛勤劳作后，他自己的设计得到了迅猛发展，同时还编撰了一本创立了架构风格的书籍。

Jefferson不仅细心研究了Palladio的书，还考察了巴黎他住所周围的建筑。[1]

巴赫花费了6个月的无薪假期，步行250英里来研究Buxtehude的成果与想法（由于请假时间太长，他还丢掉了工作）。事实证明，巴赫是个比Buxtehude更为出色的作曲家，但他这种优秀却是来自于对前辈所用技术的研究与使用，并非对其视而不见。[2]

我认为优秀的技术设计者也需要这么做，但现代设计匆忙的节奏却对这种实践非常不利。

除了每个设计者都需要做的以外，旨在做出优秀设计的技术设计原则需要具备可供访问的范本以及对它们的真知灼见。

计算机与软件设计呢

在计算机与软件领域中，基于范本的设计处于何种状态呢？坦率地说，我认为我们都远远落后于传统的设计原则。对于使用范本的设计来说，我们都有很棒的技术与资源。但很多课程对其的重视程度还远远不够，对设计实践的渗透还不是很高。

你使用何种范本

在传统的设计原则中，业余设计者与经过训练的专业人士对于范本的使用是大不相同的。

业余选手会使用在自己的经历中所见过的那些范本。专业选手的可选范围则要大很多：整个范本库反映了不同时代、不同风格、不同思想流派。这样，他就能够从这些范本库中得到专家级的指引，老师会重点指出值得关注的特征并说明它们的差异。

计算机与软件领域中的很多设计都表明设计者只会使用个人经历中曾经见到过的范本，甚至连专业选手都不会学习现有的范本。

计算机。计算机架构反映出了架构师一开始进行大量编程所用的机器所带来的深刻影响。因此，早期的DEC微型电脑非常喜欢使用MIT Whirlwind；IBM 704与1401则大量使用了IBM System/360；毫无疑问，早期微型电脑的灵感来自于DEC PDP-11。

人们还能够看到公司的历史——与建筑不同，设计计算机的架构师就工作在实现计算机的公司里。经过有针对性的培训以及日常文化的影响，他们要比其他竞争对手更熟悉公司早先的机器。Intel微处理器就非常明显地反映出了特定公司的风格。

在这个领域中，通过改编进行设计是很常见的事情。成功的计算机会繁衍出众多兼容的同辈与后继者，这常常是通过向早先模型中添加功能来实现的。

批量生产的软件。像是Microsoft Word这样的产品仅仅跟随计算机的设计模式，后续产品会不断修改功能与实现。Lehman与Belady对这种方式进行了仔细研究并记录了下来。[3]

客户化应用软件与操作系统。从历史上来看，大多数客户化应用软件与操作系统都能反映出设计者的经历而非整个原则。

最近，文档与模式教学为这个领域提供了可相互借鉴的能力。Gamma (1995)的《Design Patterns》一书擅长数据结构与组件级别的模式；Buschmann (1996)的《Pattern-Oriented

Software Architecture》一书解决的是大范围、系统性结构模式。我们需要了解关于整个系统的更多描述和解释系统概念。总的来说，只有很少的操作系统做到了这一点。

学习范本的设计原理

设计者该如何学习所在领域中的范本呢？要想学习某种架构，你可以阅读手册。要想学习某种实现，你可以阅读维护文档。但总的来说，你必须学习关于产品的技术文档与书籍才能搞清楚其基本原理。

然而，大多数技术文档都在强调什么（what），但却忽略了为什么（why）。很多设计者从来都没有说清楚他们的设计，创建者都在忙着接下来的设计呢。

早期的任何技术与随后的变革中都充满了异常情况，解决方法也是五花八门，争议之声从来都没有停止过。这些文档就像军方的胜利报告一样总是事后诸葛亮，他们通常具有各种借口；也就是说，他们对于回顾来说是合理的，但对于实际的设计过程来说却并非这样。对于我们来说，这种过程充满了凹坑、死胡同、错误的转向以及不断改变的目标。我们从这些异常情况到事后的抚平中学到了很多。

计算机处理器架构为范本的学习提供了大量的示例。这种技术是最近才出现的，有很多值得说明的地方。该领域一开始充斥着各种各样的设计方法，但最终却聚焦于“标准架构”上。Blaauw与我在《Computer Architecture》（1997）一书的第9章详细说明了这种演进。演进——虚拟内存、迷你电脑、微型计算机与RISC架构——都在历史的发展进程中留下了浓墨重彩的一笔。每一种技术都引起了争论，这也促使其不断丰富自己的基本原理。

第一代计算机

关于它的最重要的一篇计算机论文是：Burks、Goldstine与von Neumann (1946), “Preliminary discussion of the logical design of an electronic computing instrument”。

这是一项令人难以置信的工作——是每个计算机科学家的必读之作。它中肯地提出了程序存储的概念，3个寄存器运算部件，还有其他很多真知灼见。这篇论文的涵盖范围非常广；言之有物，值得一读。

Maurice Wilkes提及了更早的一份草案：

我熬夜阅读这份报告，立刻发现这才是我们真正需要的东西，从那时起，没人再去怀疑计算机发展之路了。[4]

Wilkes进一步提到，这篇论文所提出的观点诞生于宾夕法尼亚大学，是在与Presper Eckert、John Mauchly与John von Neumann之间的讨论中得出的。他后悔人们常常将这个硕果记在von Neumann一人身上，要想纠正这个错误的观点是很难的。

“初步讨论”之后，很多地方的小组都开始使用真空管逻辑电路构建存储程序机了。首个成果诞生于曼彻斯特，出现了可运行但使用困难的雏形，在剑桥出现了首个可用的存储程序

机——EDSAC。他们的基本原理都被详细记录在案：Williams (1948), "Electronic digital computers", Wilkes (1949), "The EDSAC"。

最重要的早期超级计算机是IBM Stretch与Control Data CDC 6600。Buchholz (1962), 《Planning a Computer System: Project Stretch》诞生出了大多数阐述基本原理的论文。然而，最值得关注的是第17章，它介绍了一种全新的计算机——用于密码分析的数据流协处理器——但对机器特性却几乎没有什么说明与原理性信息。

CDC 6600很快就超越了Stretch成为世界上最快的计算机，并且统治了科学计算超级计算机领域。它是Cray家族超级计算机的前身。Thornton (1970), 《The Design of a Computer—The CDC 6600》一书阐述了CDC 6600的大量原理。

第三代计算机

第二代计算机架构很快就被抛弃；因为它们缺少足够的地址位来处理越来越便宜且必不可少的更大内存。很多产品线架构不可避免地出现了不兼容的情况，这令人感到非常痛苦。幸运的是，集成电路在实现代价上进行了巨大的改进，高层次的语言也可以重新编译，这样就可以转换到新的架构上了。新的架构需要新的原理。

Blaauw与Brooks (1997), 《Computer Architecture》虽说不是一本讲述原理的书，但却包含了很多System/360架构决策的原理。我们可以根据自己掌握的知识来解释它们。Amdahl (1964)与Blaauw (1964)给出了System/360原理的简短对照表。

虚拟内存

Manchester Atlas从较慢的后备存储器向更小的高速内存的转换过程中引入了自动化的指令块分页机制。位于密歇根与MIT的分时操作系统开发者们不久之后就提出了借助于更大的命名空间将这个概念应用到完整的虚拟内存中。GE与IBM构建了这种计算机。这又是一场革命；全新的原理：Sumner (1962)(Atlas), Dennis (1965), Arden (1966)。

小型计算机的变革

晶体管——二极管逻辑电路提供了全新的、更加便宜的计算机实现方式。DEC PDP-8就是这类机器，它使得独立部门而非整个机构就能承担并控制计算机的制造任务，这极大地改变了世界。这种社会进步至少能与技术上性价比的进步相提并论。小型计算机得以批量制造，它们与所谓的大型主机共存而非替代掉它们。

大型主机制造商们很满足于他们的业务模型——赚得盆盈钵满，感到很开心——他们基本上都错失了这场小型计算机变革。很多新的计算机制造商开始出现。这其中最成功的当属Digital Equipment Corporation（DEC公司）。Bell (1978)阐述了DEC微型计算机的原理与变革。

微型计算机与RISC的变革

类似的社会与技术变革也出现在了集成电路领域。更低的价格意味着个人而非单位就可以

拥有并控制他们自己的机器了。微型计算机得以更大批量地制造出来。

这次轮到那些成功过的小型计算机制造商们赚得盆满钵满，感到很开心了。他们错过了这场微型计算机变革。Hewlett-Packard（HP公司）存活了下来，而DEC则没能幸免于难。一些大型主机制造商，尤其是IBM又重新登上了舞台，成为主要的个人微型计算机制造商。

这场革命又一次诞生出了大量原理：Hoff (1972)(one-chip CPU), Patterson (1981) (RISC I), Radin (1982) (IBM 801)。[5]

其他学科的专家能够轻松列出类似的列表，将这些记录下来的范本赋予历史的进程、变革以及里程碑。

如何训练才能改进基于范本的设计

如果设计者需要掌握整个工艺的想法与技术，但这些内容已经超出了他们自己以及其他人的经验，那么这种工艺该如何发挥作用呢？

范例集合

上一节曾说过，对于计算机架构来说有很多文档化的范本。很明显，接下来需要将这些系统化的集合装配起来并发布出去。Gordon Bell与Allen Newell在其1971年出版的图书《Computer Structures》中最先阐述了这些细节以帮助广大的设计者。Hennessy与Patterson在1990年发布了颇具价值的《Computer Architecture》，其附录E非常有用。Blaauw与我将《Computer Architecture》的第9～16章加到了集合当中。

超越集合

集合搜集完后下一步就需要仔细、公平地考察特定的范本了。在计算机设计中，我们会在关于集合的书籍与特定机器说明的评论中看到它们。

除了考察还需要进行分析，对范本进行比较，根据每个范本的目标进行评估。分析家不知不觉就会对产品的目标提出批评而非对设计满足这些目标的效力进行分析。这类分析对后面的设计者帮助并不大。

接下来需要进行比较分析了。设计的某些特性会比较好，而其他部分则可能比较弱。对于设计问题来说有些方法是可行的，有些则不行。因此，细心的分析家会从每个示例中得出最佳实践规则以指导新的问题。在大多数工程学科中，这些规则被整理成册，最终形成了标准。

软件设计怎样呢

计算机采取了集合、批评、比较分析以及综合规则的步骤而取得了长足的进步，但软件设计却被落了下来。

或许这仅仅是发展时间还不够长的问题吧。软件工程作为一门学科起始于1968年，[6]计算机工程则起始于1937年[7]。到目前为止，我们已经具有了个别操作系统的范本说明、编程语言

说明以及基本原理等。[8,9]

描述操作系统的架构要比计算机困难很多。其功能更加复杂且非常多。此外，操作Link的语义要比Divide更难以描述。我觉得我们现在所处理的复杂度已经提升了两个数量级。这当然会对集合、批评、分析以及主要软件范本的合成造成延缓。我很高兴地看到Grady Booch开始编写《Handbook of Software Architectures》了，目前这本书只在网上有，还没有印刷成纸质书籍。

谁？为了学习的目的将范本系统化是一门学问而非设计问题。学者与设计者在品味与气质上是不同的。设计者经常会将一个项目的结论用到另一个项目上，不会进行过多反思，更不必说学问了。只有成熟的学科才会吸引到学者（或是想要反思的成熟设计者）。

如何鼓励？现代工程学术界重视组织者的工作吗？人们可能这么做吗？在很多机构中，这种工作在科学史与技术部门都很有价值，但在工程部门则不然。

范本——懒惰、创意与自满

咳！上面关于设计中范本的讨论忽略掉了某些话题，这些话题是每个设计者都会遇到的：

- 难道不要复制早先的设计仅仅是因为懒惰的原因吗？诚实的专家能否完全做到这一点呢？
- 有人之所以成为设计者是因为他们喜欢创造。有趣的是，难道需要将一个人的自我表现限制在另一种风格的牢笼内吗？
- 这个世界非常重视创造与革新，需要给予他们充分的尊重，有时还有名声与财富。
- 对于人类的特殊贡献取决于人们独特的想象。忽视或压制这种创意是否会造成伤害？[11]

一些观点

由于担心会有误解，我在这里要强调一下，我并没有说过大多数设计问题都可以通过范本来解决，也不提倡依葫芦画瓢的方式。

我的观点是：

- 设计者应该很清楚范本，了解它们的优势与缺陷。创意不应该成为无知的借口。
- 工程与艺术不同，在工程当中，没有缘由的革新（也就是说，在某些情况下这种革新的效果并不“好”）是愚蠢的想法，是自私的行为——因为这不可避免地会导致意外的后果。
- 掌握了其前辈风格的设计者具备了创意的基础，他们更容易实现创意。

懒惰

当然了，懒惰或马虎的设计者可以使用范本，稍作修改就可以用了，以此简化自己的工作。总的来说，仅仅依靠复制的设计者根本不会使用年代久远的范本，他们只会使用最近、最流行的那些范本。这个世界充满了根据Frank Lloyd Wright“大草原”风格设计的懒惰的Bauhaus以

及平凡的建筑。

在任何设计领域中都不能懒惰，都需要高度的热情与勤勉才能掌握大量的范本。

创意与自满

依我来看，目前对设计创意的奖励有些偏离了方向。为了改编维特鲁威（Vitruvius），不管使用何种方法，人们只希望能够满足功能需求的设计在压力下能够保持健壮与持久的品质，并且能够给用户带来快乐。Shaker的家具、Revere的餐具以及Peck与Stowe的老虎钳都是这方面的表率。[12]

什么是创意呢？它可能是某种情趣。我们曾看到过非常新颖的设计，我们对如此优雅的解决方案感到兴奋——莱特曼可折叠的口袋工具、漂亮的玩具以及钢缆拖引的桥梁。

但情趣在于解决老问题的新方案那出众的优雅性，而不在于新颖本身。每次我们使用工具或玩具时的新情趣就足以表明这一点。它不会消退。另一方面，只有新颖是对满意的一种欺骗。7天的惊奇很快就会过去。随着新颖的退却，情趣也将一并消失。寻找新奇事物的人永远都是被动的。没有永恒不变的情趣。[13]

把创意当做目标或意外收获。寻找创意的人倾向于寻找新奇的事物而不是永恒的情趣。另一方面，致力于使设计真正发挥作用的人更倾向于具有持久价值的设计，这通常能变成意外收获。

自满。与创意之后的奋斗紧密相连的就是自满了，希望自己能够出名。长久以来，这都是人类失败的根源，它会影响到所有的设计并毁掉一切。

早在通天塔（Tower of Babel）之际它就显现出来了。“来吧，让我们建造一个通往天堂的塔楼，这样我们就能出名了。”[14]

Shelley在他的一个诗歌中揭示出了古代人与现代人的欲望：“我叫做王者之王，万王之王；看看我的作品，很强势，但却让人感到绝望。”

把欲望当做动力会让很多作品走向堕落。[15]

注释

1. Howard (2006)，《Dr. Kimball and Mr. Jefferson》。

2. Tovey (1950)，“Johann Sebastian Bach”说道：“事实上，在巴赫时代是没法得到帕莱斯特里那（Palestrina）之前的音乐分支的，我们并没有在他的手稿中发现任何样本。”

3. Lehman (1976), “A model of large program development”; Parnas(1979), “Designing software for ease of extension and contraction”。

4. Wilkes (1985)，《Memoirs of a Computer Pioneer》，108-109。

5. Hoff (1972)，“The one-chip CPU-computer or component?”；Patterson(1981), “RISC I”; Radin (1982), “The 801 minicomputer”。

6. Naur (1968), "Software engineering"。

7. Aiken (1937), "Proposed automatic calculating machine"。

8. Multics, UNIX, oS/360, Linux。

9. Sammet (1969), 《Programming Languages》; Wexelblat (1981), 《History of Programming Languages》, Bergin (1996), 《History of Programming Languages》。

10. Booch (2009), "Handbook of software architecture"。

11. Wren's St. Paul大教堂表明光荣可能来自于传统，也可能来自于创新。

我非常喜欢迪斯尼世界的风格，它也表现出了创意。想象一下灰姑娘的城堡、汤姆历险记的孤岛、鬼屋、罗宾逊的树上小屋以及19世纪的大街。在这种环境下，甚至连对风格夸张与模仿都是可行且富有情趣的。

12. Heath (1989), "Heath (1989)" 很好地概括了维特鲁维。它宣称维特鲁维提出了一种设计方法，尤其是分支方法，这使得人们能从45种房屋类型中进行选择。这是关于范本使用与简化的设计方法的主要参考资料。

13. 我认为这是对神圣的真品与邪恶的赝品的真正测试。因为真品才能带来满足（满足="足够"）。人们可以得到足够的事物、足够的睡眠、足够的娱乐、足够的爱情。但错误的方式总是在寻找新的美味、不同的感觉、不可思议的东西。

14. *Genesis* 11。

15. 我从事过这种建筑。根据其环境的要求，Sitterson Hall可以轻松构建完美的四合院并且面朝卡罗来纳旅馆。这是一个高度相同、优雅的殖民地建筑，使用同样的砖块堆砌的。"创意"使得Sitterson变成完全不同、丑陋至极的钢制屋顶，第三层的窗户太高了，落座的人根本没法欣赏到窗外的美景。它根本没有呈现出清晰的四合院美景。

读书笔记

由于空气动力的错误设计导致了塔科马港市纽约湾海峡大桥的坍塌

第14章

专业设计者缘何犯错

但当我犯错时实际上却是一件好事！

——Fiorello La Guardia

困扰专业设计者的错误不在于错误地设计了东西，而是设计了错误的东西。

错误

在任何领域中，业余选手都会犯大量的小错误，而专业选手则绝不会犯这些错误。训练、实习以及实践都会提升专业选手的技能。

当专业选手出现大的差错时会导致在修建时出现坍塌的桥梁、没有楼梯的房屋、严重浪费内存带宽的计算机、容量太大而难以学习的程序语言。

Henry Petroski建议每次材料或技术革新后，设计者都应该完成如下事情：

- 谨慎进入
- 掌握新的方法
- 开始大胆的扩展，经常忘记潜在的假设
- 大胆、自信过了头，或许由于狂妄和竞争压力感到很紧迫

他引用了一项研究成果，该研究记载了连续30年主要桥梁的坍塌情况，研究表明我们将成为下一个。结果，明尼阿波里斯市（Minneapolis）I-35W大桥的坍塌证明他是正确的。[1]

或许专业选手失败的主要原因在于新生代设计者的出现，他们从一开始就接受新技术的训练。他们没有经受过创业之痛，这经常会引起一些争议，新的专业选手通常意识不到假设与警告。

他们通常还意识不到新技术是如何适应整个技术集的。他说专业选手习惯于只见树木，不见森林，还会迟缓地问道："我所做的对于全局有意义吗？"为了了解托马斯·杰斐逊(Thomas Jefferson)，专业选手常常沉浸在正确地做事上，但他们却没有停下来问问"我做的事情是正确的吗？"

在开发System/360计算机家族时，我们经验丰富的计算机架构师所组成的小组拒绝使用自动化的内存管理，这种疏忽几乎在最后时刻才得以修补。

对于专业设计者来说，成功是一件危险的事情。失败会促使人们去分析、仔细审查以及重新思考。成功会导致设计技术与设计者过度自信。这两种信任可能会发生错位。

曾经最糟糕的计算机语言

专家也会失败的一个生动的例子就是IBM的Operating System/360 Job Control Language (JCL)，现在叫做MVS Job Control Language for z/OS。我觉得这是世界上最糟糕的计算机编程语言。它是在我的监管下开发出来的，这对于所有的管理层来说都是一种耻辱。

作为一门编程语言，JCL有很多缺陷，了解这些缺陷是有教育意义的。人们必须知道由真正的专家所组成的软件团队在随时待命，Fortran的最初设计者与主流的语言理论家在一起通力合作怎么还会犯这样的错误呢。

虽然这些错误发生在45年前，但JCL现在还在为人所用，使用方式基本上也没有发生什么变化。这些错误会继续伤害到我们，而教训则是永恒的。

何谓JCL

OS/360最初的设计目的是作为一个批处理操作系统，但从一开始，终端用户就可以通过向工作队列中发送任务、建立任务、查询状态以及获取结果来与之交互。JCL是一种脚本语言，规定了用于计算批处理任务、装载输入文件、处理每一个输出文件以及与程序和数据文件管理相关的其他大量不常用功能的选项与优先级。比如说，JCL脚本可以编辑源程序、连接到库程序、执行特定的数据集、打印、将记录存储到磁盘上以及在几种输出磁带上进行归档。

JCL非常难学，也非常难用。有一些JCL命令可以很好地控制计算过程，但很多用户会盲目地使用它们。只有一些具有探索精神的人才能深入到JCL脚本中来改变其中的内容（除了显式参数）。甚至到今天，Fortran与COBOL还是会使用附带的JCL将归档程序存储到“老式风格”的文件中。

JCL到底怎么了

JCL最大的缺陷是它实际上是一门编程语言，但其设计者却没有把它当做编程语言。

面向所有编程语言的调度语言。JCL在其深层概念上就存在着缺陷。

OS/360为大量编程语言提供了编译器，除了Fortran与COBOL外至少还有6种。每个用户都至少需要知道两种语言：JCL与它所使用的编程语言。但大多数用户都做不到这一点，因此借用的JCL处于非常尴尬的地位。

事实上，用户真正需要的并非是一门单独的调度语言（比如JCL），而是一种调度能力，就像为PL/I与S/360宏汇编程序所提供的编译期能力。这样每个程序员就可以使用单独一种语言了，以此为编译期、调度期以及运行期指定某些行为，其中大部分行为是在运行期指定的。

语法更像是S/360宏汇编而非高级语言。由于错误地决定使用一种调度语言，因此设计者选择了错误的方式。到1966年时，完整的OS/360已经上线并运行一年的时间了，汇编语言工作只占全部工作的1%左右。这时出现了一种主要的范式转移，但人们却没有意识到这一点。

但与S/360宏汇编语法还不太一样。JCL与S/360宏汇编语法还是存在很大差别的，掌握了S/360宏汇编语法并不意味着也掌握了JCL语法。

依赖于卡片列。出于某些原因，Fortran只能处理IBM 704（1956）的36位字，允许72个字符构成的语句外加上连续行。一行中第72个字符以外的字符均会被忽略（卡片的73～80列最初用于连续的数字程序卡片，这样如果将其丢弃，那么很容易重新将它们记录下来）。

JCL遵循这种穿孔卡片格式，确切时间是在将其余的OS/360作为终端访问基础之际（那时以及之后的终端甚至都没有显示出数字字符的位置，这样用户就不知道何时已经到达了第73个位置）。随后出现了主要的范式转移，这种转移是由该系统产品推动的，但人们却没有意识到这一点。

动词太少。设计者最引以为豪的地方是JCL只有6个动词：JOB、EXEC以及DD等。事实也是如此。但语言所要完成的功能数量却远不止6个。[2]强加的“优雅”简化性与任务中所固有的复杂度并不相符，这种复杂性必然会在这类临时方案中出现。

声明参数具备某些动作含义。必须要以某种方式提供动词的功能。因此JCL通过大量的关键字参数提供了一种Data Declaration（DD）语句。很多参数都假借动词之义，比如DISP，它的意思是在某个任务结束后该如何处理返回的结果集。

几乎没有分支。大多数编程语言的中心概念是条件分支。JCL却没有这种中心概念——分支是后加的，限制在某个动作中，通过参数来实现。

没有迭代。JCL中没有提供直接、原生的方式来处理迭代；它必须使用笨拙的分支来完成。设计者在调度脚本中没有考虑到迭代动作。

没有整洁的子程序调用。类似地，设计者没有在调度脚本中体会到对子程序调用的需求。对于众多的JCL程序员来说，要想大量使用子程序，只能重复使用相同的命令序列（只有某些参数不同），这着实让人无法理解。

JCL缘何是这样的

设计JCL的专家们将他们太多的经验带了进来。他们的故步自封阻碍了进一步的思考，从

这个角度来看，遵循范本反而成为了一场灾难。

OS/360 JCL的主要设计者来到了OS/360项目中，该项目来自于大获成功的IBM 1410/7010操作系统（1963）。就功能来说，1410/7010 OS要比OS/360简单两个数量级。它是个严格的批处理操作系统，主要用于传统的文件维护应用，但却没有远程处理功能。调度功能（比如文件名与I/O设备分配）是通过一些简单的控制卡片实现的，这些卡片放在每个任务的板子前，它们会被传输到读卡机中，这种技术可以追溯到磁带操作系统时期。

OS/360 JCL的设计者们将他们的任务视为对1410/7010设计经验的一种回顾，因为他们说过“使用一些控制卡片来处理调度”。这是个致命的错误。这个目标说明的每一部分在概念上都是错误的，这种错误导致整个设计都充斥了错误的想法。

没有几种控制卡片具备1410/7010操作系统的特性，少数人把简单性当做OS/360 JCL的目标。 这导致了过少的动词类型。不仅过少的卡片类型是错误的，每一种任务都应该由一些卡片进行控制的想法也是错误的。结果，JCL脚本通常会包含大量语句。

卡片是第二个概念上的误导。整个JCL编程语言在概念上是构建在穿孔卡片上的，但那时穿孔卡片行将过时。

控制卡片表明每一个卡片都是单独解释的，行为上几乎是独立的——事实上，这是早期操作系统的做法。它占据了这门糟糕的编程语言的有限分支、迭代以及子程序设施。

将卡片当做单独的完整命令说明了为何没有人认识到JCL会变成一门编程语言——一门在调度期解释并执行的语言的原因。我们基本的问题是愿景缺乏想像力。

随后的问题是设计者一直都没有真正设计好JCL——它只不过在不断成长而已。虽然曾被当做是一门系统语言，但根据语言设计者的专业知识与经验，本可以将它设计成为一门语言。

然而事实却并非如此。一开始，作为“一些控制卡片的设计”这种思想干扰了设计任务调度器这个主要的任务。文件系统管理任务、远程处理网络管理等在OS/360设计期间也在不断发展，每个新的调度功能或是规范都被放到了JCL中。因为语言的灵活性、普适性以及综合结构性太差，新的规范只能以新的关键字参数的形式添加进来，大多数都是DD语句形式。因此，声明中本来应该是形容词的最后变成了命令动词，这影响到了所有的动作。

小结

1）失败的例子要比成功的例子更值得我们细心研究。

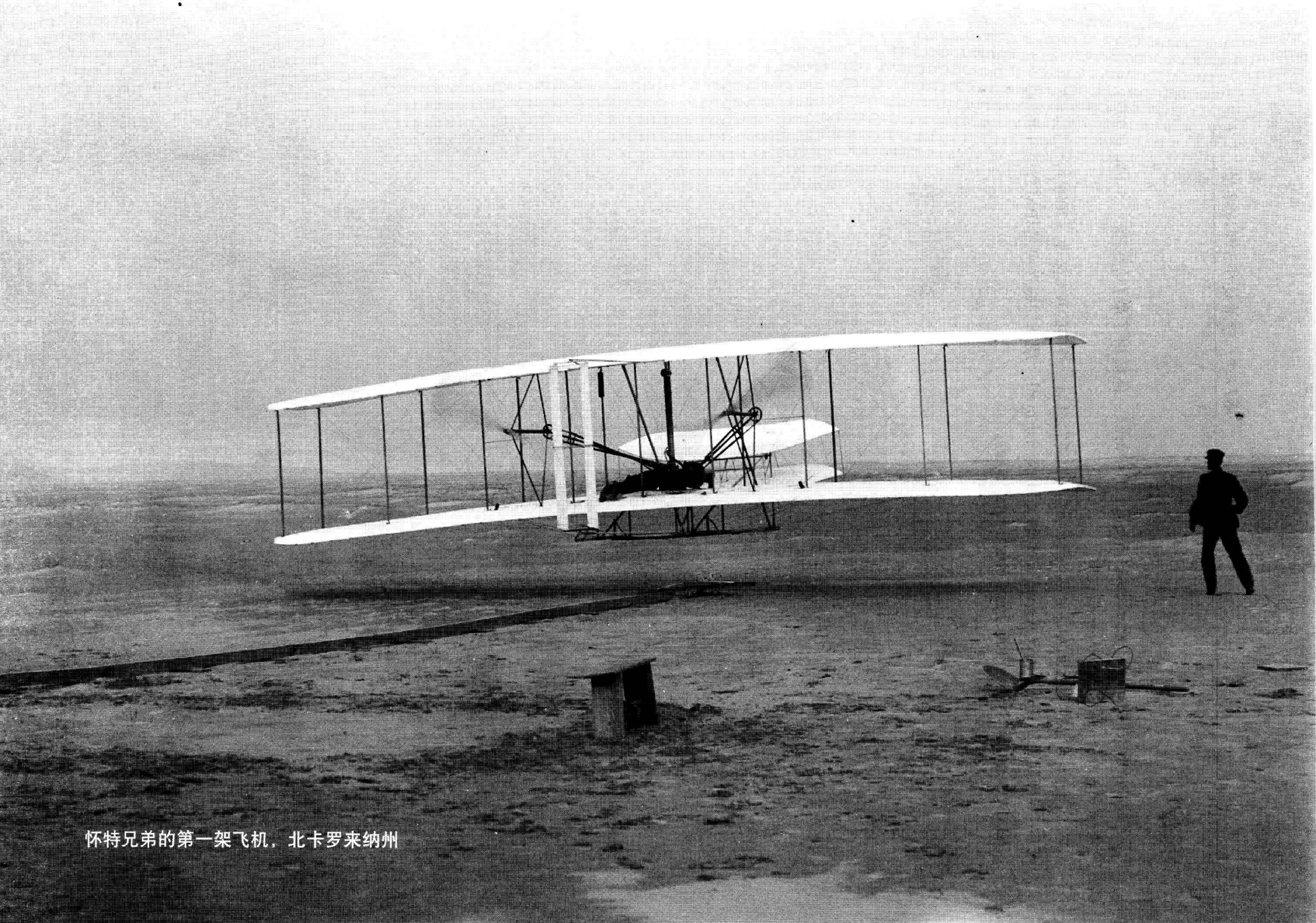
2）成功后要审视自己。对于设计技术、设计本身以及我们自身来说，成功会激励我们自信。但这一切可能会导致过度自信。

3）从上层角度思考你的设计目标以及对目标所处的周边环境的假设。是否处于范式转移中呢？你的假设在十年后还有效吗？你是在设计正确的东西吗？

注释

1. Petroski (2008)，《Success through Failure》。他援引了最初的研究，Sibly (1977)，"Structural accidents and their causes"。

2. 随着JCL的不断发展，更多的动词被添加到它里面。当前的JCL标准（2008年11月）可以在下列网址查看：http://www.isc.ucsb.edu/tsg/jcl.html。最初的标准可见IBM公司(1965)，《IBM Operating System/360, Job Control Language》。



怀特兄弟的第一架飞机，北卡罗来纳州

第15章

设计的分离

大约在16世纪，在大多数欧洲语言中出现了“设计”这个术语或等价的词……总的来说，这个术语表明设计从实现中分离出来。

——Michael Cooley (1988)

设计与使用和实现的分离

20世纪在设计原则方面最大的进展之一就是设计者与实现者和用户的进一步分离。

请考虑19世纪到20世纪之交的那些发明家。爱迪生在他的实验室里制作了他的所有发明的能工作的实物。亨利·福特造出了他自己的车。怀特兄弟亲手造出了他们的飞机。

一个世纪之后，哪个计算机工程师能够造出他自己的芯片？更不必说从沙子和铜开始了。哪个飞机设计师能够掌握飞机的复杂制造过程？更不必说动态稳定飞机的软件了。哪个建筑设计师完成了自己的结构工程并针对地震进行加固？

类似地，在许多学科中设计者也与用户分离了。在建筑方面，医院、火葬场、核燃料处理工厂、生物物理学实验室的设计者很少有作为用户的个人经验，必须通过用户代表提取出预期用户的行为。也许更糟，设计者要与用户的辅助人员打交道，这些人与真正的用户之间还有一段距离。基本没有哪个海军设计师指挥过一艘舰船，更不必说指挥舰船作战了。

这与一个世纪之前的情况形成了强烈对比。今天的汽车是由高级工程师设计的，他们10多岁的时候在众所周知的绿荫树下拆分老汽车。今天的高级通信专家大多数都有业余无线电执照，可能在学校中装配过单管收音机。今天的一些英国高级机械工程师是1-3-1“三明治”计划的产物：1年在公司的亲手操作培训，公司出资在大学学习3年，再加上1年手把手的培训，然后再开始设计。许多美国的工程师是协作培训计划的产物，结合了大学的知识和手把手的行业经验。

幸运的是，这种分离仍有例外。例如，软件工程还是一门年轻的学科，系统架构师们曾经做过程序员。个人产品的设计者总是其用户，如iPod、iPhone和汽车等，他们自己的使用感觉激发了他们的设计灵感。UNIX的设计者，特别是开源Linux的设计者，都是从他们自己的需要

出发，构建工具给他们自己使用，并与其他人分离。我认为这对成功的使用和用户的激情都是有益的。

为什么分离

第一个理由很明显。20世纪在所有实现技术上的惊人进步要求专业化和更长的学习时间。跟上地震工程的进展，甚至仅仅是利用复合材料进行制造，这在目前都是一项全职的工作。

第二个理由不太明显，但可能同样重要。我们现在设计的东西如此复杂，以致光是它们的设计都要求专业化、更长的学习时间，以及所有设计者的努力。现在基本上没有不复杂的技术。以简单的奶油蛋糕的复杂制造过程为例，在考虑好口味的同时，还要考虑较长的保质期，以及馅料和蛋糕始终保持分离。[1]

分离的结果

结果怎样？我们可以看到什么后果？沟通问题很多。建筑师建造了优雅的建筑，在其中工作却很难。工程师设计的控制面板，让核反应堆的运营人员觉得很困惑。过度指定的实现使成本超出了本来应有的成本，增加的功能或性能却不多。对于用户与设计者的联系、设计者与实现者的联系来说，他们之间的沟通带宽大大减小了。人与人之间的沟通总是比一个人自己沟通要差得多。灾难性的、高成本的、让人尴尬的沟通错误比比皆是。

补救措施

首先一点，设计者必须意识到，这种分离在20世纪已经发生，必须投入额外的努力和专门的工作，以缓解它们带来的痛苦效应。

补救措施1：用户场景体验

即使是少量的用户场景体验，也比没有要好。即使是好的用户体验模拟，也比没有要好。全尺寸的实物模拟让我们能够尝试厨房或座舱的场景。虚拟现实的环境也能起到这种效果。

当我受命为IBM Stretch计算机设计一个操作员控制台时，我只是道听途说操作员实际上在做什么事，根本不知道他们的一些任务的相关频率和重要性。

Stretch团队在夏天停工两周，大部分成员都去度假了。所以我来到为波基普西实验室运营709计算机的计算机中心，申请做2周的操作员学徒。

这个过程提供了大量的信息。我的主要工作是装上磁带，但是我完全沉浸在科学计算中心的工作节奏中，敏锐地观察那些主要操作员所做的事情。[2]

这段“用户”体验导致第一个操作员控制台的设计是程序控制的（基本上是一个紧密连接

的终端），而不是直接反映并影响硬件，它能够为多名操作员提供多个终端，在多名程序员之间灵活地分配任务，并能够在线交互式调试程序。

我必须承认，我设计的这个过于有想像力的控制台似乎很少以我设想的方式在Stretch的安装上使用。在线交互式调式相当长一段时间以后才实现，部分原因是因为Ted Codd的Stretch多程序操作系统只是选件，不是标准Stretch软件。[3]

在我参与Operating System/360的设计时，这段经历就更有价值了。在线交互式调试的所有要素都具备了，以前的探索导致了更精巧的终端和充分的软件支持。

Dave和Jane Richardson在杜克大学的生物化学实验室的学期休假也是类似的体验。每天的接触帮助我理解了他们对模块化图形工具的需求，他们利用这些工具来研究蛋白质的结构和功能。

Philippe Kruchten在担任加拿大航空控制系统的首席架构师时将这种接触系统化了：

所有软件人员都送去进行航空控制的手把手培训，参加ATC（航空控制）课程，然后花上数天时间，在真正的区域控制中心里，坐在控制人员旁边，尝试理解他们的活动的本质。编程使用Ada，达到的效果是使他们有足够的共同语言，以便能够有效地协同工作，利用彼此的技能。[4]

补救措施2：通过增量式设计和增量式交付与用户密切交互

Harlan Mills的增量式开发和迭代式交付体系，是从项目一开始就与用户保持密切接触的最佳方式。[5]先构建一个能工作的最小功能版本，然后让用户使用，或至少以测试来驱动开发。即使是为大众市场构建的产品，也可以通过部分用户采样的方式进行测试。

在我自己为科学家构建交互式图形系统工具的实践中，我经常惊讶于用户对我们早期原型系统的反应。我几乎会完全错误估计他们使用新工具的方式。

我们的团队花了大约10年的时间才实现梦想的“room-filling蛋白质”的虚拟图像。我当初的想法是，知道了C端和N端的物理位置，化学家们就可以更容易地在复杂的分子中找到研究途径。在经历许多次失败之后，我们终于在头戴式显示器上产生了一张合适的高分辨率图像。化学家可以边走边看蛋白质的结构，研究感兴趣的部分。

我们的第一个用户来赴她两周一次的例会，一切都很顺利，她移动得不少。第二次会议也一样。第三次会议时她说：“我能坐下吗？”10年的工作被一句话打败了！和体力上的消耗相比，这种导航帮助不值。

我们在一个辐射治疗规划系统上也有类似的经验。辐射专家的任务是找出多个粒子束的方向，它们将轰击肿瘤，并避免像眼睛这样的敏感器官。我们将病人的半透明虚拟人体放在空间中，这样医生就可以走来走去，从不同的角度来查看。不是这样的，他们更喜欢坐着，让虚拟的病人旋转各种角度。

补救措施3：并发工程

设计者需要投入更多精力，深入到实际的体验和实现过程中去。即使是单独的、不具代表性的实现经验，也可以很好地告诉设计者，实现的某个版本太过于理想化或不完整。我强烈建议这样做。

有一种危险，那就是如果设计者的个人经验就是全部输入（这从本质上就是不具代表性的），朴素的样例实现经验将对设计产生过度的影响。也许最好的平衡是在主要设计实践中采用并发工程。在这种情况下，真正的实现者积极参与设计过程，他们的丰富经验为设计者有限的实现样例提供了平衡。（在软件领域，同样的实践有时候就被称为敏捷方法。）

将实现前推到设计过程也提出了一些自己的要求。习惯按照标准工程图纸来工作的船厂工人，可能不太习惯于通过标准平面图和局部图想象出完成后的结构，因此就不能够抓住错误或预见到实现的“陷阱”。通过丰富的视图来增强标准平面图和局部图，甚至通过虚拟现实的环境来察看，可能使并发设计过程更顺畅。

补救措施4：设计者的教育

设计课程必须包括理解用户需求和期望的技术与实践。[6]

Gould和Lewis在经典的、影响深远的1985年的论文中阐明了3个设计原则，排在第一位的就是通过“从开始就直接联系用户”来理解用户及其任务。他们发现许多设计者认为自己在这样做，实际上是在听别人说用户如何如何，阅读有关用户的材料，分析用户的特点描述，只在开发过程较晚的阶段才向用户“展示”、“复查”或“验证”设计。[7]

在车间的工位上的实现经验，真真正正地构建软件，在设计者教育中是非常重要的。

学生对直接用户接触和实际实现经验的需要就要求更多的项目课程和体验，即使是牺牲一些书本知识的学习也是值得的。分析技术和正式的综合方法是必要的工具，但高级的方法在需要时就会不言自明。Gut直觉是很难获得的。今天的设计课程必须考虑到设计的分离，并付出巨大的努力让年轻的设计者接触到实现和使用的真实世界。

注释

1. Ettlinger (2007)，《Twinkie, Deconstructed》。

2. 并聆听声音。我与Grady Booch有同样的怀旧情怀：“我怀念老计算机发出的声音。通过计算机发出的声音，我就能说出我的程序在做什么。”

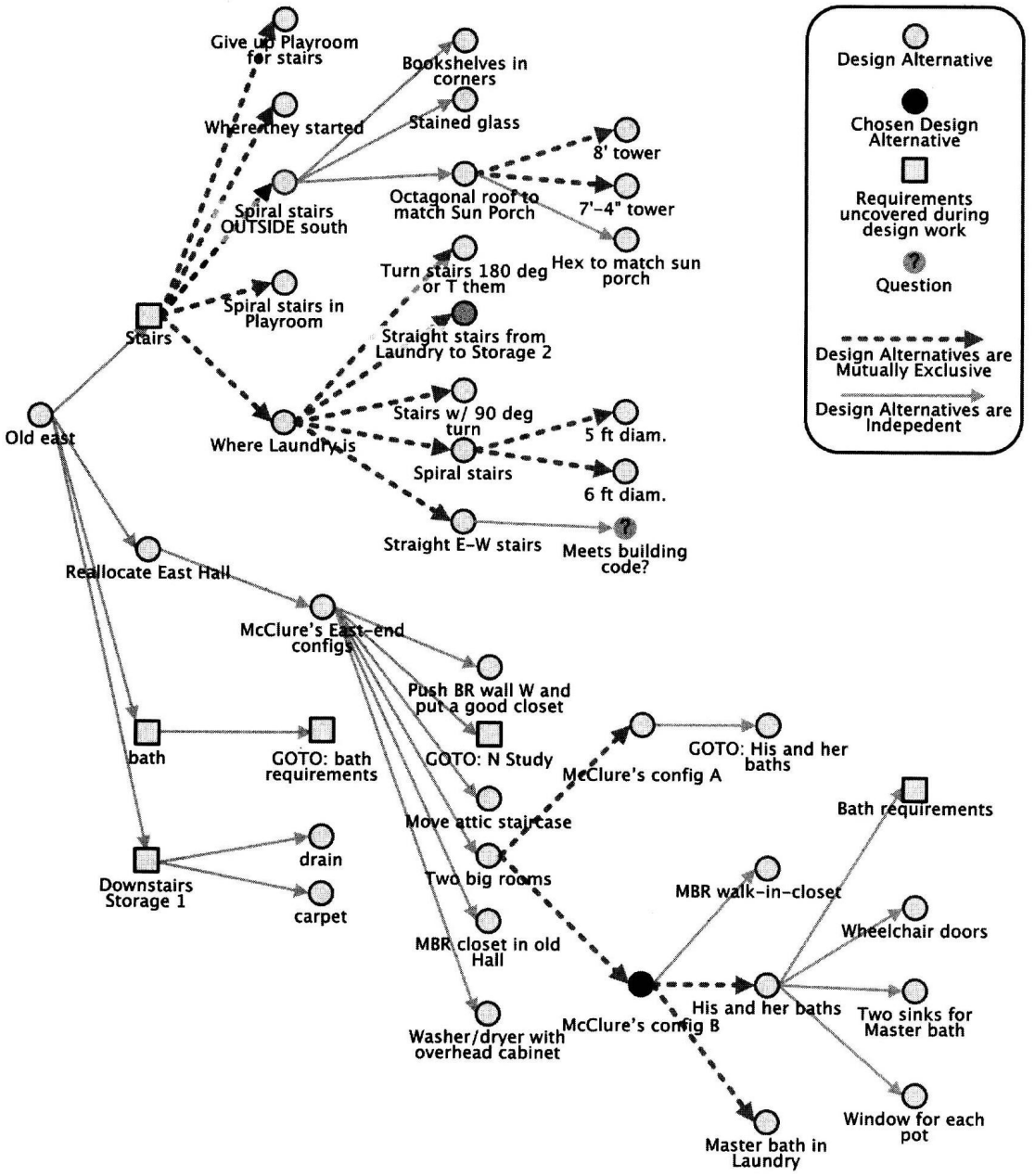
3. Codd (1959)，“Multiprogramming STRETCH”。

4. Kruchten (1999)，“The software architect and the software architecture team”。他进一步报道说：“有些人感到受挫，认为是在浪费时间，但后来很惊奇地发现这对他们的工作有很大的帮助。”

5. Mills (1971), "Top-down programming in large systems"。

6. 在大约22个软件工程实验室课程中，我发现有必要也有可能请一些外部用户，学生团队必须与他们一起工作，满足他们的要求。用户必须每周花一些时间与开发团队开会，他们的回报就是可能得到可用的原型软件。我要求的项目是那些成功了就会有用，但不一定必需的项目。学生团队必须允许失败。

7. Gould (1985), "Designing for usability"："这些原则是：尽早关注用户并持续关注用户；对使用进行经验性测量；迭代式设计"。



Brooks厢房设计的Compendium图的局部

第16章

展现设计的演变途径和理由

与Sharif Razzaque合作

利用一台数字计算机，可以有许多种方法让你自己变成一个傻子，再加一台计算机也不会有什么不一样。

——Maurice Wilkes爵士 (1959)，《The EDSAC》

在修复你不懂的东西时，要小心。

简介

设计师要想从每次设计经历中学到最多的东西，他们就需要记录下设计的演进过程：不仅说明设计是什么，而且说明设计为什么会变成这样。同时，这样的设计理由文档对系统维护者的帮助也非常大，它防止了许多无知之错。记录下设计和演变途径和理由，这比初看起来要难得多。[1]

一些研究团体尝试用计算机辅助解决这一过程中的问题。[2] 所以，Sharif Razzaque和我决定，针对一个特定项目的设计演变途径，开发一个计算机展示。我们以235页的散文式日志作为原材料，这些日志是我妻子Nancy和我在设计1 700平方英尺的屋子扩展时保存下来的。（这个设计项目在第22章中进行了简单介绍。该项目更大的设计树发布在本书的网站上。）

本章展示了我们对真实设计演变途径和设计演变文档的观点。这里我们的描述方式如下：

- “我们”指我们一起完成的工作。
- “我”指Razzaque独自完成的工作。
- “我”指Razzaque的解释和假设。
- “我们”指Brooks夫妇同意这些解释和假设。

知识网线性化

当Vannevar Bush为他建议的Memex系统的设计而致谢时，所有知识项的相互关系需要一张图来表示，一般来说是一张非平面图。[3]

但是这样的图难以展现，也几乎不可能理解。所以各行各业的人们将知识的表示线性化，并以一种或多种辅助的表现形式作为补充。

具体过程是：

1）切断图中的一些边，直到它成为一棵树。这一过程强加了一种以前没有的层次化顺序，不论这种顺序是否是想要的。

2）将这棵树以某种熟知的方式映射为一条线，通常以广度优先的方式实现。

以本书为例。它所面对的主题是相互纠缠的。但这本书本身必须是线性的：一页接着一页，一行接着一行，一个字接着一个字。所以作者将主题内容组织成一棵树，并在内容表上显示它：小节构成章，主题构成小节。页码显示了从树到线性形式的映射关系。

但是，内容表并不是故事的全部。在书的后面可以有索引，它以字母顺序的术语来组织这本书。某个术语的页码实际上定义了贯穿全书的一条链子。索引恢复了我们将网状内容转为树状内容表时切断的许多链接。

组织一个图书馆时也采用了同样的过程。图会图书馆分类系统（或称杜威十进分类系统，Dewey Decimal System）将所有相互联系的书映射成一棵树。树再通过广度优先的方式映射为一条线，对应到书架顺序。但这种映射以多种索引作为补充，每种索引都恢复了一些切断的链接：作者索引、标题索引、主题索引。

主题索引特别有趣，因为书架顺序映射已经是基于主要的主题了。根据除主要主题之外的其他主题，主题索引重新组织了这些书。

维基百科通过丰富的链接来解决这种网状的表现形式，马上就能访问。这种能力是对我们的智能工具箱的重要增强。

任何一种设计空间都有类似的网状结构，所以设计的表现形式具有挑战性。如果说设计都难以有效地表示出来，那么设计过程更是天生如此。

第2章中介绍的理性设计模型似乎假定存在这样一棵设计树，它在每个分支节点展示了选择所带来的次级设计决定。在理想情况下，人们会将每个选择的理由与这个决定联系起来。但决定是以多种复杂的方式相互关联的，每个决定既有其简单原因，也受到它的兄弟节点或远亲节点的影响。

我们的设计演变途径记录

我们的目标是记录Brooks家厢房设计所隐含的设计树，这既是第22章中简要描述的设计过

程案例的补充，也是为了展示随时间推移的设计演变。更重要的是，我们希望深刻理解Brooks家的设计过程：

- 日志与Fred的回忆一致程度如何？
- 有哪些争执，它们发生在什么地方？
- 突破何时发生，如何发生？
- Fred和Nancy是否系统地探索过设计树？
- 从这次分析中得到的发现是否支持本书其他部分的观点？

结果表明，我们在尝试重建设计树的过程中所学到的，比设计树本身揭示了更多的东西。实际上，树本身只带来了少数领悟，这有些让人失望。这项练习是一次失败的实验。

我们研究房屋设计过程的过程

我们从寻找已有的设计树绘制软件开始。最终我们选择了Compendium，[4]这个工具现在主要的用途是在设计过程展开时记录并关注设计的过程。

我以日志第一页中的笔记作为设计树的根，一页一页地推进，根据我事先准备好的转换方案，将笔记转变成结点和链接。我们很快就遇到了困难，迫使我们考虑转换是否正确。这导致我修改了转换方案，偏离了Compendium隐含的方针。

我们的过程形成了一种模式：每次我们调整转换方案时，我就会回到第一页，对Compendium树进行返工，以符合新的转换方案。然后我们会推进，并不可避免地遇到另一条日志记录，它不适合我们的转换方案。这导致我们重新审视：

- 我们（不断演进）的转换方案；
- 我们使用Compendium重建设计树的方法；
- 设计过程本身。

这个过程（转换方案遇到困难，调整方案，重新开始）一次又一次发生，最初是每天都会发生，后来发生的频率越来越低。我们逐渐收敛到一个较好的方案，并决定对余下的缺点妥协，只有这样我们才能在日志转换为设计树的工作上取得进展。

什么是设计树

不久后我就意识到，我们在为设计树寻找转换方案上遇到的问题，源自于缺少对设计树的准确定义。我头脑中的定义是非正式的、隐式的、模糊的。寻找一个可用的树转换方案的过程，也是寻找设计树定义的过程，这个定义需要足够严谨、全面和精确，以便在操作上可行。

因为我的定义是非正式的和隐式的，所以我甚至都没想起来要抛开软件工具，用纸和笔一个设计树的例子，并针对它确定一个转换方案。

我们开始模糊的设计树概念与图2-1中的设计树相符。这种概念就像人们在配置一个按订

单生产的笔记本电脑时遇到的选择树一样。每个设计问题（也就是要做出的决定）都是一个节点。兄弟设计问题，如“可见性”和“闹铃”，相互之间是正交的，设计者必须回答每个问题。在Blaauw (1997)看来，这称为“属性分支（attribute branches）”。

对于每个可选的设计选项，每个设计-问题节点都有一个子节点。以笔记本电脑为例，人们必须从几个可选项中选择一种显示器尺寸。这些可选项是相互排斥的可选分支。对每个独立的设计问题，设计者选择其中一个可选项。

大多数的选择带来更多的设计问题（例如，决定使用发光拨号盘之后，必须选择它的发光机制）。这些设计-问题节点是以前的解决方案节点的子节点。因此这样的设计树包括独立或互斥的设计选择，完成的产品不是以单个节点的选择来表示的，而是以许多设计选择节点构成的集合来表示的，选择的叶节点代表了每个独立设计问题的解决方案。

为了展示设计这棵树的理由，每个选择都应该与一些节点关联起来，说明它的优点和缺点。每个设计-问题节点也应该有一个关联的节点，说明所做的选择和理由。

这个关于带理由决策树的最早概念似乎很自然地符合Compendium预先定义的节点类型。我选择了这样的映射：每个设计问题用一个问题图标来表示。每个设计可选项用一个想法图标来表示。每个想法图标有优点和缺点两个子图标。选中的可选项变成一个同意图标，再带上一段理由注解。

最后，我们认为所有的设计问题，即使是独立的，也都应该根据屋子的空间按层次组织。例如，我们希望将所有起居室的问题放在“起居室”节点之下，以符合日志记录的结构和标签。

Brooks很早就将设计任务分解成了3个独立的问题（见图16-1）。

深入设计过程

设计不只是满足需求，也是发现需求

对日志一页一页的分析很快就表明，即使架构计划已经确定，需求仍然在改变。例如，建筑师Wes McClure曾建议添加公馆，一部分在北面，一部分在南面。[5] Brooks夫妇最终否决了这个想法，因为：a）这样做的结果使得屋子没有一个中心区域，让家庭成员很自然地聚集在一起；b）南面的部分需要移动宝贵的大黑橡树。但在McClure设计公馆时，没有意识到中心区域和保留橡树是需求的一部分。通过分析建议的设计解决方案，即公馆的想法，Brooks夫妇发现了这些需求。

我们在日志中再次看到了这种模式。设计工作不仅是满足需求，它还引出需求。我们的经验与第3章中熊恩（Schön）的理论产生了共鸣。好的设计过程鼓励这种现象，而不是压制它。

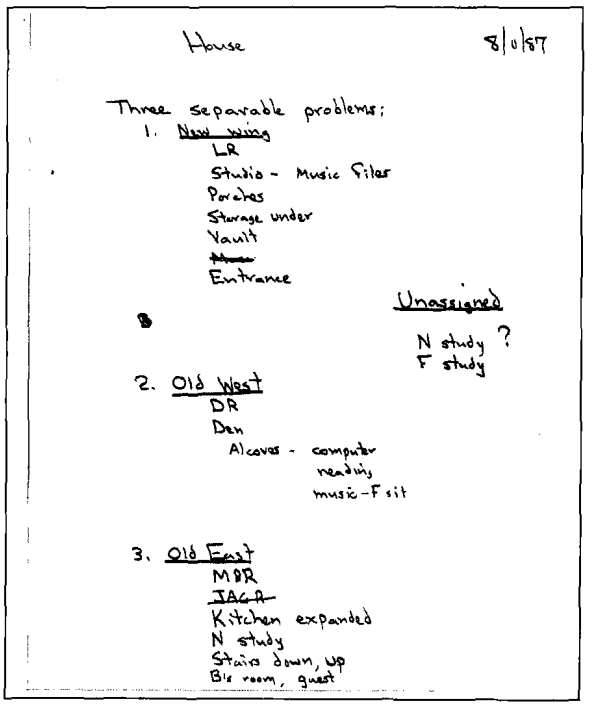


图16-1 将房间侧翼的设计任务分解为可分离的问题

设计不是简单地选择可选方案，也需要意识到存在潜在的可选方案

当设计者提出一个设计问题后，通常不可能简单地罗列出所有可能的选择。有些选择是明显的或事先存在（也许借鉴自范例）。但其他一些选择是创新的，需要突破。在日志中，我们看到Brooks夫妇为两种可能的音乐房配置而纠结，在大量的分析和几次尝试改变两种方案之后，他们认为没有一种方案可以接受。他们发现了第3种配置并立刻喜欢上它。日志中记录："配置C——真正可行之路！"

这种模式也在日志中重复出现多次。正如第3章中讲述的，对于安放音乐房的问题来说，Brooks向邻居购买土地并不是一个明显的解决方案。设计的一部分主要工作是意识到设计选项的存在。我们的经验再一次印证了熊恩的理论。

设计变化时树也变化——如何展现演进过程

这3个独立的问题看起来非常像笔记本电脑的配置树，不同之处只是最低的节点不是叶选项，而是子设计问题。但这棵树表明有些设计已经完成了。为什么入口在新的厢房？将入口设置在新的厢房，而不是在旧的西部，这本身就是一项设计决定。设计问题树已经暗含了一些设计决定。

但随着我们深入日志，就开始不太明白为什么在设计过程中，某些房间分配在一个厢房，而不是分配在另一个厢房。Brooks随后领悟到，他考虑屋子时同时想到了原来的屋子（在这个建筑项目之前）和最终建成后的屋子。例如，今天他认为Fred的书房是新厢房的节点，因为这

是建成后的结果。当他开始写第一页日志时，Fred的书房在楼下，还没有确定要移到哪里去。但在这两种状态之间还有许多不同的设计。

每一个这样的高层设计都伴随着不同的组织树结构。随着设计的改变，同样的房间和部分被放在不同的高层节点之下。例如，原来的屋子和1987年的设计将主卧室放在东厢房，但最后的设计将它放在了新厢房中。应该采用哪个树结构？我们如何展示设计随时间的演变？

我们的做法是挂上建成后屋子组织结构的所有节点，以便得到一棵结构稳定的设计树。最终的屋子设计被分成阶段I（其中又分成新厢房和旧西部）和阶段II（厨房和游戏室）。

某些设计可选项当然是设计树的高层节点。例如，“将屋子重头到底翻修一遍”肯定在阶段I和阶段II（或新厢房、旧东部和旧西部）之上。但这个节点会让读者迷惑，因为它在设计过程只占据了很短的时间。但是这样的高层节点总是事后可见的，也是强加的。

我们考虑过将这样一些短时间的探索放在一个顶层结点中，名为“早期漫游”，表明许多项目在早期探索过一些激进的设计方式。但在设计的每个阶段都会放弃一些设计可选项。那些早期放弃的设计方案实际上与后期放弃的一样，只是它们会影响设计树中的更大部分。随着设计的演进和稳定，影响深远的变更就越来越少。

决策树的结构随着时间推移而发生变化。记录下这样的变化需要新的动态工具，而这样的工具还不存在。它不仅必须追踪随着时间推移，树向叶节点方向的发展，也必须追踪节点和它们的子树从一个分枝上砍下，嫁接到另一个分枝上。

决策树与设计树

如图2-1所示，最终完成的设计（即产品）不是表示为决策树上的一个节点，而是一组叶结点。对于这样的树来说，在设计项目中的某一天，很难显示目前完成的最好设计是什么。

所有可能设计构成的空间完全是一棵不同的树，其中每个节点确定了一些产品构成的子树，类似于在这个节点下面的所有决定，并在它下面进一步分化。我们称之为设计树。每个叶节点是一个不同的完整设计。

从组合等价的角度来看，设计树比对应的决策树更大。例如，n个独立两分设计问题构成的决策树有2^n个节点。但对于任何实际的设计来说，这棵树都太大了，人们似乎不太可能构造它，这样做不能带来什么领悟。对于实际设计来说，设计树太麻烦了。

但设计树这一概念是有教益的，有助于理清思路。例如，对敏捷软件开发有一种类比：每天晚上的构建对应于设计树的一个节点，它代表目前完成的最好设计。

模块化与紧密集成的设计

在转换Brooks屋子决策树的过程中，我们发现在一个决定的可选项之间进行选择时，很少与其他选择无关。例如，音乐房可以在新厢房的北部或西部（这是一个高层决定），但它的位

置影响到书房、起居室和厨房的可能位置！

这导致了一棵不雅观的决策树，因为某些可选解决方案必须与一些属性绑在一起。例如，我们不能简单包含一个“音乐房位置”的决定，再加上一组简单的可选方案，我们必须包含以下的复合可选方案：

音乐房在西部，起居室在北部
音乐房在北部，厨房在南部
音乐房在北部，威望在北部

而且即使这样，它们也不是独立于其他设计可选项的。在《Notes on the Synthesis of Form》(1964)一书中，Alexander(1964)解释了这种紧密的依赖关系（也就是缺少模块化）是一种不利因素，因为它使得设计的修改变得困难。因此设计者不能够简单地在可选设计方案的优点之间进行选择，而是有意地、正确地在设计品质和将来修改的容易性上进行折衷。这正是Parnas(1979)在他的基础性论文“Designing software for ease of extension and contraction”中所极力主张的。[6]而且，人们也可能在设计品质和设计过程的速度和容易性上进行折衷。

模块化设计更容易表示为设计树。实际上，这可能就是我们采用模块化设计的意图。

另一方面，完全的模块化也有缺点，优化的设计有一些组件可以完成多个目标。请考虑单片式车身的汽车（unibody car）：车身不只是为了美观和装载乘客，它也是结构。单片式车身比底盘分离式设计更轻、更坚固。但与单片式车身相比，底盘分离的小卡车可以更容易地转变成SUV。

Compendium和可选工具

我们调查了一些软件包，希望找到一种软件包适合构造和分析我们这个设计决策树的例子。下面是我们的发现。

Task Architect

Task Architect[7]确实对设计有帮助，但不是我们希望的方式。Task Architect这个工具协助用户进行任务分析，即如何执行工作的结构分析。它的例子既包含了手工任务，如在汽车装配线上安装头灯，也包含了思考任务，如决定是否放弃一次着陆。

因此，对于Brooks屋子的项目来说，Task Architect可能用于更好地理解Brooks夫妇的用例（烹饪、举行会议、教授音乐等）。Task Archcitect的设计目标不是为了重构决策树，实际上也难以让它做到这一点（但我们确实尝试过）。

项目管理工具

像Task Architect一样，项目管理工具，如Mircrosoft Project、OmniPlan或SmartDraw，可能对设计系统的过程有用，但似乎不适合展现决策树。

像“项目评估与复查技术（PERT）”这样的关键路径方法是项目管理工具底层的模型。PERT似乎只支持瀑布模型。因为没有条件任务或节点，PERT技术意味着假定主要设计决定已经做出。[8]

IBIS和它的衍生品

IBIS（基于问题的信息系统）是在20世纪80年代设计的，目的是支持协作制定决策并记录下决策的理由。[9]像Compendium一样，它的设计意图是用于决策制定过程，以保持设计会议的效率，并帮助确定薄弱的逻辑。

针对我们评估的每一件工具，我勾画了一个转换方案，以符合我们对这个工具的需求。我们首先快速查看了Compendium，然后是Task Architect。当我们查看IBIS时，发现它非常自然地支持我们认为需要的许多字段和节点类型。

IBIS是一个命令行程序。Conklin的gIBIS是IBIS的图形化版本。[10] gIBIS应该比Compendium更适合我们的需要，但我们找不到一个版本能够运行在我们的计算机上。Compendium实际上是gIBIS的后代产品。所以我们回过头来采用了Compendium。

Compendium

Compendium有很多好处。它相当灵活，而且正变得越来越灵活。它有一个活跃的开发者团队，极其负责地响应请求和求助。有一个很大的用户社区，支持着一个活跃的在线论坛。因此Compendium总是在不断发展，处理新的用法。

但是，在仔细反思之后，我们不能推荐用这个工具进行设计或记录设计的演变途径和文档。

对于设计本身来说，我们担心，如果设计者在设计过程中使用一种结构化的表示法或软件工具，这将限制使用模糊的概念，妨碍概念上的设计。同样，对于创造性思想的快速探索来说，CAD工具过于精确了，草图则让设计者不用那么精确。Conklin自己也曾表示，对于设计过程的某些创造性的方面，gIBIS过于结构化且过于麻烦。[11]

对于我们重建决策树的任务来说，我不认为Compendium是最合适的软件工具。我们最终的转换方案与Compendium的目标用法相去甚远。我不再利用Compendium提供的功能来帮助重建设计树，而是在发现一些创新的方式，重塑Compendium的功能（例如利用Compendium的参考节点来描述需求）。

而且，我们的树即使在努力减小规模之后，也比Compendium用户界面设计容纳的规模要大得多。但与大多数实际的软件项目相比，这个设计任务要小得多。在这样大的树中寻找一些节点是很困难的，要打印出来或输出成图形就更困难了。

我们最终的转换方案利用了：

- 一些绕过Compendium结构的方法；
- 我们自己强加的一些结构来确保设计树的一致性，而不是Compendium的结构。

因此我相信，如果一种通用的画图工具包含树的自动布局、自动重排关系箭头、查找关联的结果，那么可能更适合记录设计树。Microsoft Visio或SmartDraw可以是这样一种选择。

DRed[12]：一个诱人的工具

关于计算机辅助的设计理由文档，我们听到的最大的成功故事就是劳斯莱斯（RR）对DRed的广泛使用。DRed是Rob Bracewell在剑桥大学的工程设计中心开发的，得到了劳斯莱斯、BAE Systems和英国工程与物理科学研究委员会（UK Engineering and Physical Sciences Research Council）的资助。[13]

DRed的目的是记录设计理由和做出的决定。它的概念结构与gIBIS非常相似。在使用方面，它看起来很像Compendium。但是因为它主要用于记录理由，DRed的演进主要集中于这一功能，这与协助设计会议的目标是有区别的。[14]

在劳斯莱斯，采用DRed是非常迅速的，因为公司已经有了很强的理由记录文化。（该项目的另一位赞助商BAE Systems没有这样的文化，他们公司就没有广泛地采用DRed。）劳斯莱斯已经要求工程师编写设计理由文字报告。管理规定的一大进步是允许项目团队使用DRed生成的文档，而不是先前要求的文字报告。DRed文档要容易得多。

根据Marco Aurisicchio的描述，劳斯莱斯对DRed的采用是大规模的。Aurisicchio是剑桥的关键人物，负责处理与劳斯莱斯的关系，他最熟悉劳斯莱斯如何使用DRed。他在劳斯莱斯讲授了许多使用课程。Michael Moss是劳斯莱斯关键人物，负责DRed的相关事宜，劳斯莱斯的工程师如果需要直接支持就会去找他。他也负责过滤反馈信息并排列优先级，报告给剑桥的Bracewell团队。根据Brooks的观点，这种用户和构建者之间的全职两人链接关系对DRed的成功起到了主要作用。

各个团队使用DRed的方式不同，但通常的做法是在设计会议上将DRed的内容画在白板上。然后指派一个人将白板上记录下的草图变成正式的DRed文档。DRed文档也会在个人设计过程中产生。它既用于概念设计，也用于详细设计。设计者、复查者和下游的制造工程师自己就能使用DRed，并不需要协助者。

所有劳斯莱斯的工程师中，大约有30%至少接受了使用DRed的短期培训，总共大约600人，他们分布在世界各地的一些部门和实验室中。新的工程师在为期6周的项目课程中学习劳斯莱斯的工程实践，4人一组。典型的项目就是一个真正的问题，某些团队希望解决它，却没有人力来做。但这不是一个十分重要项目，所以培训学员允许失败。[15]

剑桥团队看到过的最大的DRed树由190张图构成，每张图平均有15个节点。对于这样的项目也会有一张概要图，其中一个节点代表着其他更详细的图。

当然，劳斯莱斯的设计会演进。他们的DRed图和其他文档随着设计的演进而演进。在多次复查过程中，DRed图对于展示者和复查者都很有用。DRed文档本身没有置于正式的版本控制之下。大家认为正式的劳斯莱斯版本控制非常麻烦，所以“如果DRed置于版本控制之下，

就不会有人用它了”。

对于DRed，有一种广泛的用途是开发者们从来没有想到过的，即响应世界各地现场报告的产品工程团队将DRed作为引导和记录缺陷分析的工具。“这里是引擎熄火的时间、地点和方式，以及当时记录下来的所有读数。那么，导致熄火的原因是什么？”

遗憾的是，DRed不容易得到。RR和BAE系统公司拥有其知识产权，目前它们选择保持私有。

注释

1. MacLean (1989), “Designing rationale,” 以及Tyree (2005), “Architecture decisions,” 讨论了理由记录。Moran (1996), 《Design Rationale》, 比较完整地概述了设计理由方面已发表的论文。Madison (1787), 《Notes on the Debates in the Federal Convention of 1787》是完整记录理由的一个令人吃惊的例子。Madison的文字也可以在网上看到，网址是：http://www.constitution.org/dfc/dfc_0000.htm。

2. Noble (1988), “Issue-based information systems for designv; Conklin (1988), “gIBIS”; Lee (1993), “The 1992 workshop on design rationale capture and use”; Lee (1997), “Design rationale systems”; Bracewell (2003), “A tool for capturing design rationale”; Burge (2008), “Software engineering using RATionale”。

3. Bush (1945), “That we may think” 是一篇了不起的论文，它提出了Memex系统，包含一般性的链接和个性化的链接，构成一个知识图，很像今天的互联网。其中提出的技术是原始的，但其概念具有想像力和预见性。

4. Shum (2006), “Hypermedia support for argumentation-based rationale”，具体可以参见http://compendium.open.ac.uk/institute/，2009年7月25日可访问。

5. 参见第22章中的规划。

6. Parnas (1979), “Designing software for ease of extension and contraction”，使用了一棵设计树作为其基本框架。

7. 具体可参见：http://www.taskarchitect.com/index.htm, 我在2009年7月25日访问了这一网址。

8. 这不是完全正确——根据每种选择所意味的时间进度，PERT可以帮助经理从可选设计中选择。

9. Noble (1988), “Issue-based information systems for design”。

10. Conklin (1988), “gIBIS”。

11. Conklin (1988), “gIBIS,” 324～325。

12. 这一部分基于Brooks在2008年7月19日对Bracewell和Aurisicchio的联合访谈。Brooks看到了系统能力的全面展示。

13. Bracewell (2003), “A tool for capturing design rationale”。

14. Aurisicchio等(2007), “Evaluation of how DRed design rationale is interpreted”。

15. UNC-Chapel Hill的软件工程项目课程完全采用真实用户选择项目的同样判据。

第四部分

一套计算机科学家进行房屋设计的梦想系统

UNC GRIP分子图形系统用户控制名James Lipscomb

第17章

计算机科学家的建筑设计理想系统——从思维到机器

金字塔、大教堂以及火箭之所以存在，并不是拜几何、结构理论或者热力学所赐，而是因为这是它们的创造者灵光一闪的产物。

——Eugene Ferguson (1992)，《Engineering and the Mind's Eye》

挑战

假设我们可以有这样一个完美的计算机系统，用来设计房屋架构以及其他的建筑。那么，它将会是什么样子呢？

一个设想

一个职业的建筑师对于整个的“建筑设计”系统的设想显然远胜于我。不过，因为我曾做过一些业余的房屋设计，而建筑架构比起软件架构来说又更为具体和更易理解，再加上我在人机交互方面超过50年的经验，在这里我就斗胆提出我眼中的建筑设计理想系统，设计师-计算机这一接口的雏形。

这一接口很大程度上是我和我的学生在参与UNC-Chapel Hill GRIP分子图形系统过程中所吸取经验教训的总结。这一实时交互图形系统是我们与那些天才的蛋白质化学家共同合作，经过数年的不断改进而开发出的应用于挑战性任务的工具。[1]

这篇文章仅限于房屋功能性设计的过程，不牵涉结构或系统工程的部分，尽管我认为同样的系统对于结构工程、力学及电力系统，甚至是家具内饰来说都是适用的。同时，也有其他理想的设计系统存在，Ullman(1962) 的一篇文章就描述了一个用于机械设计的系统。[2]

然而我们所讨论的范围截然不同。Ullman追寻的是一个能最大可能处理知识管理的程序，这是一个很有意义的目标。而我在此所考虑的范畴是设计者思维与自动化系统之间交流，在我看来设计者的思维才是最为重要的。

渐进完善

良好设计是自顶向下的。正如写作时先是拟定提纲而后再发散内容，编程时首先要抽象出数据结构和算法一样，那么设计住宅的时候也是先识别出基于使用体验的功能性空间，而后才是它们之间的连接性。建筑的审美亦是如此，由大众的角度作为切入。

顶尖的设计者，就算是最敢于破除陈规的先锋派们，也很少是从零开始的——他们也是站在前人的肩膀上。[3] 他们借鉴贯通各家所长，融汇成概念完整而风格内聚自成一体的设计。

常用的技巧犹如国画大师画画一样，宣纸铺洒的时候已是成竹在胸，先作出骨架，而后再不断加入更多的细节进行完善，使骨架精致丰满起来。

Turner Whitted在1986年提出了另一种可能是更为优越的构建物理对象模型的方法（起源于计算机图形学领域）。一开始仅构建了模型所需的部分，接下来，一个属性一个属性地进行修正，将结果不断地接近新事物在脑中的设想，或者现实世界中的实际属性。

Whitted将这一技巧称为渐进完善。[4] 从十分现实的层面上来说，渐进完善正是几个世纪以来自然科学所采用的程序，用这种建模方式来模拟现实中的自然生物。[5]

就人为设计的工件而言，其设计的过程会引起所要达到的理想目标发生变化。渐进完善这一手段非常有帮助。在设计每一步过程中都有一个原型可以揣摩。最初这一原型就是有效的，不存在结构上的不一致。该原型是详尽的，无论从视觉上还是听觉上都不会有理解的偏差。

因此我们可以假设这样的一个住宅设计步骤：

- “给我一个带三个卧室的乔治亚风格住宅。”
- “坐南朝北。”
- “从左向右镜像翻转。”
- “起居室要有14英尺宽。”
- “将厨房的纵深缩短一英尺。”
- “使用白水泥面的而不是砖面的外墙面。”
- “使用波形瓦而不是砾瓦的屋顶。”

我认为Whitted的设想十分有说服力，我将以此来设定我的理想系统。这可能对如何进行设计产生影响。新的工具将引领我们以更合理的方式进行思考。

模型库

这一理想系统是以一系列良好而详尽的设计范本库为基础的。从这些风格一致的范例着手，除非设计者本身的失误，否则一致性可以得到良好的保证。模型库本身随着系统的使用也在逐渐地发展。为了进行模型的改造，需要用于捕获3D物体并约化为结构模型的计算机视觉技术。

在我们最初设想的住宅声明步骤中，设计师表现为对于范本库了然于胸：通过名字调用范本。

不管设计师经验多么丰富，随着范本库的增长，这种简明的熟悉程度很快就会消失。就如同你对一个地域的了解程度，由于太过于广阔，有些部分对你来说就像自家的后院一样熟悉，有一些部分也还过得去，而对于其他部分来说你则是像一个外来者一样的陌生了。因此，对范本库进行有层次的扫描这种能力在此显得格外的关键。

对范本库进行结构化的处理是一项分类学的工作，但实际上很多术语和概念结构已经存在了。[6]

渐进完善模式的不足

尽管我假定最具生产效率而便于使用的设计系统应当遵循渐进完善的原则来构建，但这一模式本身也有其内在的危害。

有人指出过于广泛地使用样例将会潜在的对设计师的创造性产生限制。像布鲁内列斯基、勒·柯布西耶、盖瑞、高迪所设计的那些作品，有可能出自被填鸭式灌输的设计师的思想当中吗？

我认为有可能。没有谁是外行人。都是通过学习先例接受训练成长进来的。就像巴赫这一类人，他们都是在精通熟练的情况下进行创新，而不是无中生有。摩西奶奶那样的人在这个世界上是屈指可数的。

更确切地说，这些“又如何”的样例记录的是一个个设计片段，这是一个很好的工具，也没必要提供延续性。

渐进完善模式真正的危害在于范本库本身。拙劣的样本、太少的模型、过于狭隘的范围——这些不足之处都将极大地限制呈现出来的设计。这一危害在一开始的时候更为糟糕。

从思维到机器输入的设想

不管是由范例开始还是从头设计，需要考虑的是如何将思考的东西转换为计算机模型。

人们想要把自己幻想中的城堡变为现实，使用语音、双手、头脑和双脚。Buxton与他在Alias Systems的同事和多伦多大学的学生一起，在双手交互人机界面方面作出了突破性的进展。[7]惯用手提供精准的操纵，非惯用手提供上下切换（简单地说，比如下一步是左或右）。两只手一起提供了相近的维度——“非常广阔。”

名词-动词组合

在设计语言里，比如命令式语言，从总体上每个陈述有一个动词和一个名词，即对象。这个名词可能有一个可选的表达“哪个”的形容词、短语或从句。而这个动词可能有一个状词短语（见图17-1）。语言学上一个有趣的特点是许多动词将形容词赋给对象名词：“将门做成32英寸宽”、“把西面的墙涂成绿色”。

在空间设计语言中，通常希望用手进行指示的方式来说明名词。而语音是表达动词的最自

然的设备；视窗-图标-菜单-指标（WIMP）接口使用了一种非自然的输入——菜单。事实上，菜单对显示选项来说有着非常多的好处。然而，对于在各种熟悉的任务间切换的用户而言，这是不必要的。

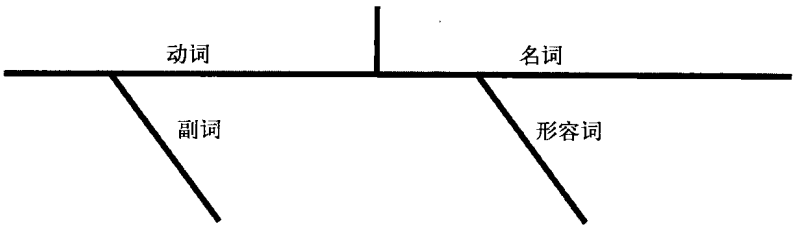


图17-1　命令语言的结构

使用WIMP接口工作的用户采用常规的组合方式：点击一个或键盘输入一个名词规范；然后点击一个菜单命令（动词）。这样，在编辑文档时，可以选择一个文本块，单击“剪切”，然后选择另一个，单击“复制”，再选择一个间距的空间，单击“粘贴”。

说明动词

单手的节奏事实上让人烦恼并且会带来反效果。将光标从名词域移到动词菜单就丢掉了之前的位置。下一个名词通常靠近前一个名词——这又需要将光标移回原来的地方。为了免于这种迷惑，对一些高频率的动词开发了一些特殊的规范。双击意思味着“打开”这个动词；键入意味着“插入”。最常见的动词都有相应的键盘快捷键，由左手执行。这一技巧可以说是天才的发明。

新手总是可以选择标准的技巧——从菜单进行动词选择。而专家则具备更快的技巧，利用双手，将光标位置留在动词域。最好的一点是，新手可以每次掌握一个动词的高级技巧，基于他自己的使用频率。

语音命令。语音发出命令是自然的模式。因此我们的理想系统需要一个有限词汇的语音识别系统，并且能容忍较大的语音误差，有着丰富的同义词组以及用户可编辑的词典。菜单选项仍是可用的，包括键盘。因此声音沙哑的用户也不会有什么损失。

泛化动词。现今的建筑CAD系统包含了丰富的泛化动词集，这来自于多年的客户经验积累。也许这些动词过多了，但如果用户选择自己定制的调色盘并从中欣然选取，使用体验就不会受到影响。

这些例子包括：

- 旋转
- 复制
- 分组
- 捕捉

- 对齐
- 空间分离
- 比例缩放
- 从库中选择对象
- 命名

从库中选择对象可能是其中最有用的，不管你是否采用渐进完善的方案。我们采用了这一方案，在库选择的频率和重要性方面都得到了提升。

库中的对象都是按同一比例尺；频率高的动词是选择和缩放，缩放自动被设置为现在使用的比例尺。

说明名词

我们在空间和连续时间上用来指定对象和区域的方式真是多种多样啊！

通过名字。在大多数对话中我们都是通过显式的名字或者是一个代词所暗指的名字来指定对象。在理想系统中我们也希望能这样。就算有百分之百有效的语音识别能力，这也比看起来要困难得多。“左边这个”指的是哪一个呢？“前面一个”？“红色的这个”？“最大的这个”？“它”？有时候就算指定的点没有歧义，但选择的范围常常不是这样。要让系统像三岁小孩一样的聪明和自然需要句法分析、语义分析，以及保存和使用上下文。

更复杂的是，同一个对象通过不同的名字调用，或者不同的对象通过同一个名字调用。要引用一个陆海空军的联合数据库就是一个无比浩大的工程。光是“现在什么时间”就是巨大的挑战。空军使用的是格林尼治时间；陆军使用本地时间；每艘海军舰只使用其所属航空母舰战斗群所在位置的当地时间。

因此我们的理想系统必须支持单独的用户和用户组来创建单独的同义词词典以辅助或覆盖系统的词典。合理的说明模型库的命名法对于我们构建系统的人来说犹如是一个沼泽，用户必须拥有和使用这些工具。

通过二维指定。WIMP接口的极大成功验证了使用指点的方式来选择定义好的和可见对象的威力。事实上这一模式确实会有非常高的使用频率。

但这还不够。当我们构建GRIP系统时，我假设第一位化学家客户将会通过指示的方式从一个蛋白质剩余物的几百个胺酸中选择一个。然而，他想要一个键盘，通过三位数的剩余物数字来指定一个剩余物。他与这个特定的蛋白质打交道多年了，他对于这些剩余物的数字名字的熟悉程度就像熟悉自己孩子的名字一样。指示要求调整视角，直到目标剩余物在三维的视野里变得可见。另外，拿着激光笔的手也会感到劳累。

通过2D素描。建筑师设计3D的实体。但他们的主要模式是2D绘图，包括严谨的绘画和草图素描。尽管最佳的2D投影也不过是3D对象的有限描绘，但这种手段却是事实，描草图对于

思考过程来说显得非常关键。[8]

不管3D建模工具发展得有多么丰富和方便，我也不认为会撼动到2D绘图的重要地位。人的视网膜是2D的。能方便的引导双手的平面也是2D的。因此，我们的理想系统必须具备2D指定和草图的特性，就像笔和板一样，不仅感知到位置还要感知到压力。

对于定义好的2D空间，比如地图或蓝图，人们通常需要指出“哪里”和“多大的区间”。有层次性的级别划分比如州、国家或者房间使得指出详细的规格变得很容易。

精确的指定“哪里”和“多大”需要双手的操作。考虑描绘一条精确指定长度的线条。右手执笔确定“哪里”，而左手操作数字小键盘来确定“多长”。

3D空间的指点和草描。以上所述关于2D的考量都可以应用于3D。指示由其内在决定更让人劳累——你必须举起手。手指-手腕-手肘的运动有着与手臂-手肘-肩膀运动同样的精确性，所以人们希望工作量的大多数时间里双肘都可以支撑。[9]

指定主观的空间区间及其旋转比起指定对象和它们的位置来说要困难得多。在对话中我们通常是经由双手的运动来指定主观的区间：“云的形状应该像这样”。这正是我们的理想系统的工作之道。大部分工作都已经通过小部件及3D规格的预示性（affordance）完成了。[10]大部分的工作旨在产出解放双手的规格，而其中的大部分不过是提供了更笨拙的替代而已。

说明文字

大多数的文字块会是短的字符串，一般是设计绘图上的名字或尺寸范围。语音识别是对其进行选择的工具，动词也是。

规格说明将会由文字块组成，这是选自数据库的标准段落参数化后的结果。要创建新的实质的文本，听写显然不行，因为一个人打字并编辑段落比起听写编辑来要快很多。对于这两类的文字工作，需要具有字母数字的键盘。

在我们的UNC GRIP系统当中，我们发现从工作界面中移走键盘的好处。在实践当中，它通常一直存在，我希望对于理想系统来说也是如此。

说明助词

“移动。”

“什么方向？多远?到哪里？”

“旋转。”

“什么方向？多远？”

“复制。”

“多少？什么方向？如何分隔？”

“选择门并且缩放。”

“多宽？”

命令对话的组成不仅包括动词，还包括名词。大多数动词有助动词作修饰，通常是前置的短语。[11]

大多数的助动词会是量化的。而且，还必须是精确的，在纸片上指点远远不够。这种精确性是通过辅助动词间接指定：扣合位、对齐，等等。

大部分是通过从有限的菜单中选择而来：调色盘、素材列表和完工表。菜单选择认知度高并且经济实惠。住宅设计，如同计算机内存访问一样，表现出很强的局部性——对于任意给定的设计决策，可能有着许多不同的独立的替代选项，然而大多数的选择都是从一个较小的子集中选择的。个性化定制的菜单非常关键。

剩下的量化精确性由数字键盘给出了最佳的指定。在我们的经验中，这得到了广泛的应用。就像字母数字键盘一样，爱不释手。

说明视点和视图

大多数创造性的建筑作品都是由平面图和剖面图完成的，但人们通过观察整体的3D设计来检查作品，并且有着许多优势，包括进行建筑的虚拟漫游。指定现有的3D住宅设计视图对于动词+助词的规格来说是一个相当重要的特别案例。

有一些视图参数会持续改变，而有些则改变不太频繁。这种区别与动态vs静态的区别不同。人们甚至需要那些不太频繁的变更的参数也能动态而平滑的改变。

内部视图

在模拟住宅的漫游的时候，x和y坐标及头部偏航角（yaw）会持续的变化。人们通过在建筑楼层的2D描绘上滑动视角来在同一平面上转动视线。

楼层上的视线高度（eye height）很少变动。通常会移动到不同楼层的同一高度。极少的情况下，人们会调整视线高度来适合不同的设计者或是不同的用户所想象的视角。

滚转角（roll）极少发生变化。我们不会大量转动我们的头部。固定的运动要常见得多——向下看、向上看——但它们比起x与y和偏航角的运动来说就少得多了。

EyeBall。我们认为一种特殊的I/O设备对于指定建筑的内部视图十分理想。EyeBall由一个底部切掉半英寸的撞球所架着的六自由度的追踪器所组成。它装备了两个可由食指和中指按压的按钮（见图17-2）。你可以将其滑过作品表面作V型的光标运动。

这一装置支持自然连续的六个视图参数规格——视点（3个参数）、视向（2个参数）、视图翻滚（1个参数）——大量使用了（x，y）和偏航解。食指的按钮是一个单键的鼠标，另一个按钮是离合器。通过升高EyeBall按离合，将视点移动到更高的楼层，而降低EyeBall松开离合

就向下移动。视线高度保持不变。一个良好的特性是视点在z轴上连续移动，就像在玻璃电梯里一样，不存在视觉的中断，所以不会在模型中失去焦点。

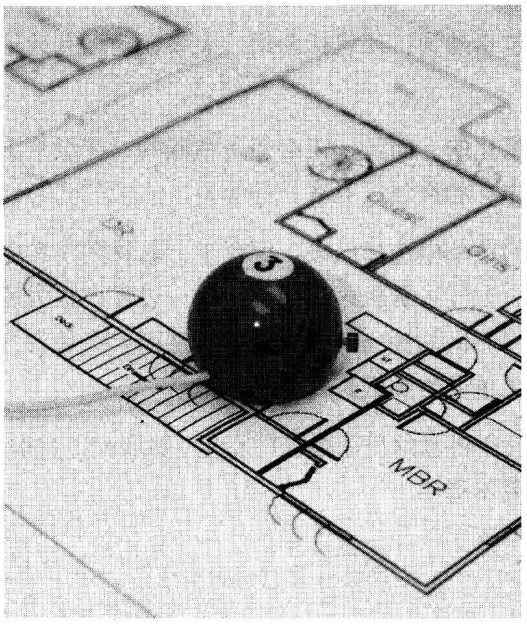


图17-2　北卡罗来纳大学教堂山分校EyeBall视图指示设备

EyeBall除了滑动平面指定内部视图以外，还有另一个模式。假设你的工作台的3D模型始终有一个像上下文索引坐标一样的镭射视点，可以指明“你在这个位置”。那么，你可以方便地指定到那个索引坐标并发出指令，“将我移动那个位置，从那个视角进行观察。”[12]

不管在哪个模式下，视线高度都是由滑动器设定的，这种情况通常很难得。两按钮对于百分之五的女性或百分之九十五的男性都是默认的。

外部视图

“牙签”视点。就如同我在第18章中将介绍的那样，任何一个设计工作台都需要所设计对象的多角度的视图：所操纵的工件以及上下文的视图。

对于住宅而言，一个上下文视图可能是从外部的视图，或是对于整个房间的视图，或者内部墙的视图。对于指定一个对象任意的外部视图，我们认为最有用的设备是二自由度的位置控制杆，如图17-3所示。

这非常易于操控，通过是使用左手从你所查看的对象来指定方向。止动位有一个“牙签”指向用户。要给出一个4p球面度的视图，我们双倍移动牙签的止动位角度。这样，当它完全移动左边或右边，操作的视图就移动该对象的正背后。出乎我的意料的是，我们的用户认为这种加倍非常容易和自然。这与经常使用的指定立体正视图和一些三维视图（从对齐的围绕的立方体的一角）默认按钮互为补充，还有一个不太常用的滑动器来指定视图距离。

图 17-3

深度感知。密集度，填充或空容积内的相对位置，对于建筑审美来说是一个重要的考虑。要认知它需要通过3D的感知。最强大的深度信号是动力学深度效应，眼睛和场景的相对运动。这一信号比立体观测还要强大，而两者的结合更是强大无比。

那么该如何指定所想要达到的运动呢？显然EyeBall可以用于任意的视点移动，但这需要更多的工作。人们倾向于一边思考一边研究密集度。在我们的UNC-Chapel Hill GRIP分子图形系统中，我们发现摇动整个场景的垂直轴能极大地帮助认识，因此我们提供了一个摇动模式，在不采取任何行动的时候场景可以像扭曲的钟摆一样运动。用户可以指定摇动的幅度和周期，虽然通常都是使用的默认值。

注释

1. Brooks（1995），《人月神话》，194。

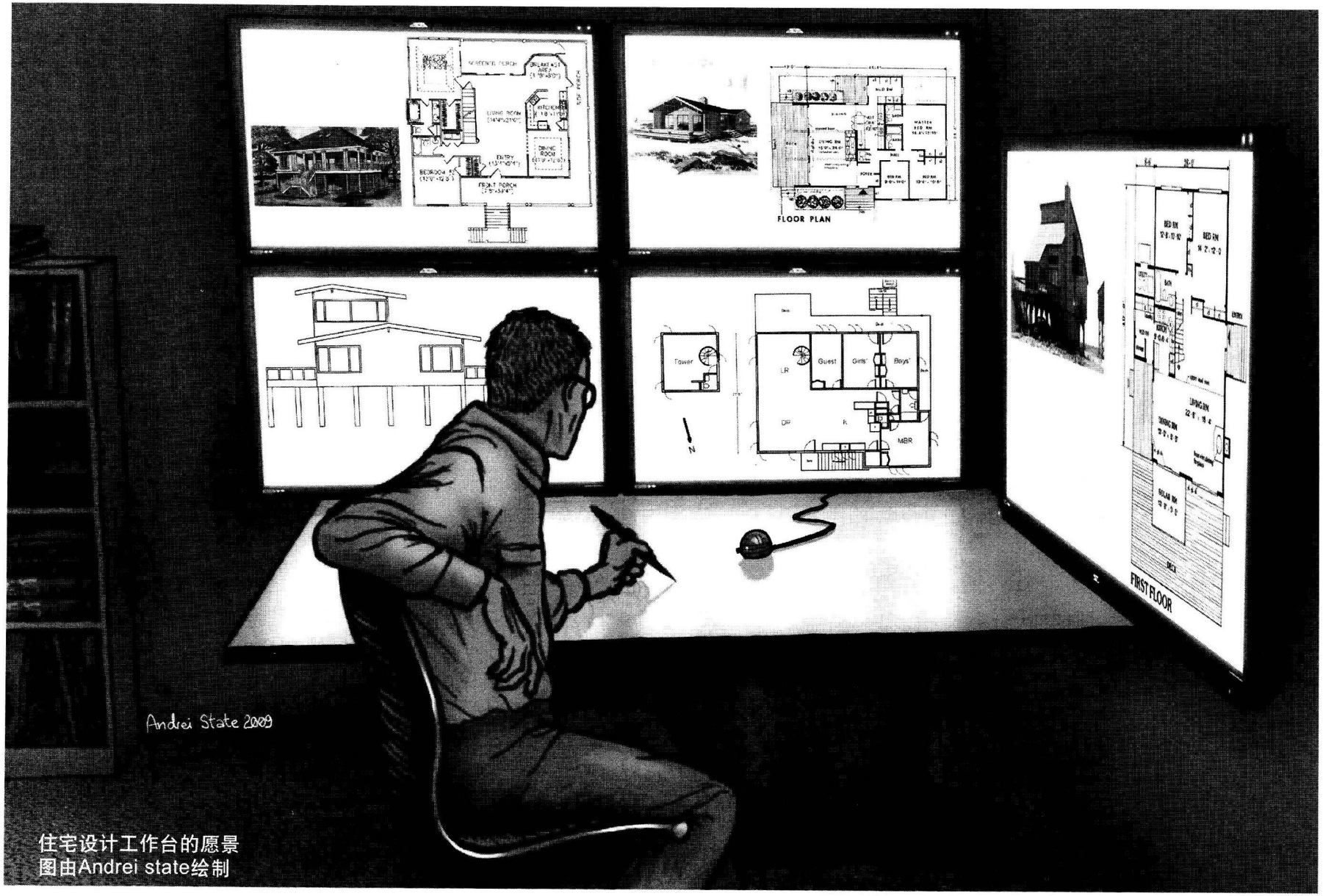
2. 参见http://products.construction.com/。

3. 系统内部的复杂性要求双向的连接，也包括进行文字编辑这样的简单情况。这一点可以从Ken Brooks的Lilac系统中领悟到，可以在“A two-view document editor with user-definable document structure”(1988)和“A two-view document editor”(1991)这两篇论文中看到。其难点在于其中一个视图比另一个承载了更多的信息。所以对简单的视频进行修改就需要向复杂的视图删除或增加信息。

4. Svensson和Kristiansen（2002），“Computational modelling and simulation of acoustic spaces”。

5. Meehan（2002），“Physiological measures of presence in stressful virtual environments”。

6. Raymond（2001），《The Cathedral and the Bazaar》第4章。



住宅设计工作台的愿景
图由Andrei state绘制

第18章

计算机科学家的建筑设计理想系统——从机器到思维

当我们准备建筑房屋的时候，
我们首先要测量地基，然后设计图样；
做好图样以后，
我们还要估计建筑的费用；
要是费用超过了我们的财力，
就必须把图样重新绘制，设法减省一些人工，
或是从一开始就放弃这一建筑计划。

——威廉·莎士比亚(1598),《亨利四世·下篇》

双向通道

思维-机器的协作要求双向的通道。连接思维的宽带通道是通过眼睛。然而，这并不是唯一通道。耳朵对于状况认知、监控警报、感受环境变化以及语音演说等有着特别的优势。触觉（感觉）和嗅觉看起来似乎能连接到更深层次的意识。我们的语言对于暗示这种深度有着丰富的隐喻。我们对于复杂的认知情况有一定的“感觉”，因为我们觉得“不太对劲”，所以需要“着手处理”。

视觉显示——多并发窗口

使用计算机的设计师通常使用一个活动窗口。然而计算机科学家长期以来都了解设计者需要至少两个窗口，而且屏幕尺寸也太小了。[1] 而我们的住宅设计理想系统到底需要什么样的视觉显示呢？

制图桌和绘图视图

由于我相信2D绘图始终会是实际设计当中的主流运用，第一个显示台是一个电子制图桌。

- **角度。**在标准的手工制图中，垂直显示和倾斜的工作表面是结合在一起的，但电子版本

使它们获得了自由。将两者分开意味着手和手臂不会再影响视图。研究表明用户在与另一显示表面相关的表面上移动鼠标或笔不再有困难。

- **工作表面尺寸**。制图桌各不相同，但30英寸×48英寸比较普遍。这些维度是由手臂的触及范围来确定的。
- **显示分辨率**。应当适合于人类的眼睛，即一个弧分（minute of arc）——1 920像素的电脑显示屏置于两倍于其宽度的距离。这样的平面显示屏如今在许多的办公室都能很好地显示。
- **显示视图距离**。投影到6到8英尺以外的屏幕上可以缓解设计师的视疲劳。

桌面上的对象几乎都是2D的绘图。屏幕上的对象或许是绘图、其他的透视图等。分层次的显示绘图并且保持可控制的透明性，如同现在常规所做的那样，是重点关注某一方面而又保持概念性上下文的一种不可或缺的技巧。

2D内容视图

在我们的UNC系统中，我们观察了用户使用它来做实际的设计工作，不管是建筑、分子学或者论文的2D图表，我们发现了一些普遍的步骤：

1）研究一个较大的立体内容区块。
2）变焦放大。
3）创建或操作某些本地部分。
4）变焦缩小。
5）重复操作。

显然，由于一个时间共享窗口的历史局限造成了这种不自然和浪费的行为。用户同时需要目录视图和详细视图，两者之间只需要眼神的切换而不需要手动。我们必须提供这一点。此外，将目录窗口实现为在详细视图某个角落的缩略图也是没有帮助的——用户也需要查看目录的详细内容。在现今的价格条件下，没有哪个认真的设计工作室可以有任何借口说显示功能有所约束。始终都应该用两个大的显示屏来显示这两个视图。

通常，目录视图是立体的，包括整个制图。但是像从库中选择对象这样的操作，一个视图可能是库（一个层次结构的树形展示或者是某个我的特点节点的目录），而另一个视图仍然是该对象所处位置的绘图。

3D视图

人们在三维的住宅里居住和活动，而不是存在于二维的抽象中。在理想系统当中，设计师持续地观察当前所指定的三维住宅——始终保持全部的细节。

投影虚拟环境技术看似非常适合于此。人们不需要一个完全的沉浸虚拟环境，比如CAVE，一个小的垂直圆屋顶就非常棒了。就算是一个单一的标准3D视图窗口也会工作得非常好。设计师显然是面对图纸的表面，而不是漫游其中。他拥有一系列方便的控制方式。这种3D显示是一个辅助的创作工作，而CAVE是为观察而设计的，不是为创作。

这里存在一些技术问题。首先，人们不想在设计时总是戴着立体眼镜。即使没有它眼睛的负担都已经够重了。所以人们需要模式的转换——也许可以是一个脚踏开关，又或者使用自由立体的3D显示。

其次，住宅通常都有平面的墙，可以作为投影屏幕。当然，人们都想要viewpointer（一种3D技术），理想的情况是用一个EyeBall来控制标准的模型，偏航角的转动让场景围绕观看者旋转，或者将二者反过来。但这一理想系统同样还需要一种捕捉模式来将视图与平面住宅的表面对齐。

外部视图

从外部查看住宅是非常有用的视图，特别是有开阔的视野。

一个印象深刻的外部视图是在夜晚的时候当屋内灯火通明时留下的视图。这种效果，画家Thomas Kinkade夸奖它具有一种迷人而又有吸引力和温暖的感觉。

我发现第三个出乎意料有用的外部视图是——近处的墙明显地被忽略的夜晚视图。在这种模式下环绕整个建筑可以得到一个非常全面的印象（见图18-1）。

图18-1　一所公寓的剖面视图

Delta Sphere公司

工作手册视图

另一个并行显示的窗口是设计师的工作手册。设计师带着这些东西来到工作台：

- 进行中的设计
- 行动计划

产出的成果是：

- 更新后的进行中的设计以及未来的行动计划。
- 捕获所有行动的日志记录。它们以及版本控制系统能够支持自动的回溯。
- 笔记，理想化的记录以解放双手进行设计，比如尝试过哪些方法，为什么，哪些被排除掉了，为什么，哪些又保留下来了，为什么。

这些为什么，虽然不能被行动日志捕捉下来，但对于任何复杂设计来说却是至关重要的。

在中断之后它们能为恢复提供很好的帮助。

在探索设计的不同可行方案的时候，这些为什么能够提供可能被忽略掉的一些其他的分支。对于团队的新成员和项目的接替者来说，这些为什么也是相当有价值的。

在理想的情况下，行动计划和结果笔记应当在一个文档里交错，通过不同的颜色或字体来区分。

在理想系统中，在页角插入了两个行动日志片段，展现当前的成本预算价值和其他的一些预算的商品，比如平方英尺。

规格视图

建造不仅需要绘图，还需要散文体的规格。最好的情况是这应当与绘图同步进行并完成了，而不是在此之后完成。

如果使用渐进完善的设计方式，这并不像乍一看那么难。规格是高度程式化的。如果库里的每个初始模型已经有相应的规格，那么设计者可以在过程中修改这些规格。这一工作由于Sweets File（现在叫做Sweets Network）大量的收集而变得极其简单。[2] 该网站收集、罗列、配图、描述并详细说明了上百万的产品。McGraw-Hill建造负责维护这些文件和其分类，产品供应商提供内容，建筑设计师和承包商订阅服务。各得其所。

所有理想系统需要第四个2D的窗口，不断提炼的散文体的规格。理想的情况是，在这一显示中所发生的变更能够自动地反映到绘图显示上。由于许多Sweets Network产品描述已经以标准的形式包括了CAD模型，这一联系从概念上理解将并不困难。而从绘图将变更传播到规格却是极其困难的——这也是目前研究的一个难题。[3]

声音展示

许多年以前我曾被一盘录影带所震惊，一群赫尔辛基的建筑师用计算机图形模拟展示了一个提议的改造计划。图像还行，但绝非特别优秀。然后这盘录影收录了背景的声音。尽管看不到人，但舞台背后的声音也让整个场景跃然纸上。

交付音频显示很容易，而构建可用于展示的声音模型却是一个挑战。人们想要建立户内的和户外的声音源：电视、交通、洗衣机、小孩玩耍和争论等。接下来，虚拟住宅漫游的时候，人们想要听到这些结果，发现其带来的乐趣和问题。

必需的原声模拟技术已经尽在掌握。[4] 现在还不能实时实现的是近似和不断改进的显示。处理器频率的飞速发展可以解决这些困难。

挑战来自门和窗户。哪些是打开的？有多远？其组合数不胜数。设计师规格和设计师勘测从原理上来看易行，但从实践上来说非常乏味。一个明显的探测帮助是绘出声音密集的结构图，以耳朵的高度，以格子标记于整个住宅制图上，对应所交互的开放和关闭的小孔。这一视图显示可以建议应当收听哪些地方。EyeBall仍然是一个方便的装置，可以用于指定模型中收听的位置和方向。

触觉展示

触觉显示似乎没有任何其他的形式更深入人心。[5] 然而，尽管我试过，但是我并不认为有什么站得住脚的理由来说服我将现有的触觉技术引入理想系统。

泛化

从设计住宅的理想系统的规格出发，可以将它泛化到许多其他的领域。例如，一个构建软件的理想系统，无法从所有的3D功能受益。但它应该包括丰富的初始库、设计显示、上下文显示、工作簿显示、测试案例显示以及其他所有的相互关联的情况。

可行性

可以开始构建这一理想系统了吗？毋庸置疑，是的。所有的技术都准备好了。对于普通的单独的住宅设计它是否也是相对可以承受的呢？我相信是的，至少大公司可以投入资本，而且多名设计师可以分享这一系统。

如何去构建它呢？增量式地构建！任何一个要一口气构建这样一个表面上考虑过的系统大多数都会失败。然而不断壮大、增量式的，由真正的设计师持续试验的项目，将会成功。最困难的部分在于如何组合一个良好的初始的模型库。

们可以想象开发一个框架的学术研究项目，有着必需的标准的输入格式和描述格式，并设法获取由开源方式贡献的模型。而这正是开源模型的声望激励可以发挥的地方。[6] 我认为设计师会把他的住宅、房屋或其他项目的设计出现在这些库里不仅视为一种名声，更是一种广告，比如出现在杂志上。基于使用情况自动精选出一个核心的库不是一件难事。如果没有被查看过，那么就排除。

注释

1. Brooks（1995），《人月神话》，第265页。

2. 参见：http://products.construction.com。

3. 由于地下室系统是深度机械化的，所以它要求两路链接，但即使是在像文本编辑这样简单得多的案例，也可以从Ken Brooks的Lilac系统中捕获灵感，参见“一个带有双视图的、用户可自定义文档结构的文档编辑器”（1988）和“一个带有双视图的文档编辑器”（1991）。困难在于，其中一个视图包含比另一个视力多得多的信息，所以对这个较简单视图所做的任何修改都要求对那个较复杂视图做一些连带的信息推导和注入工作。

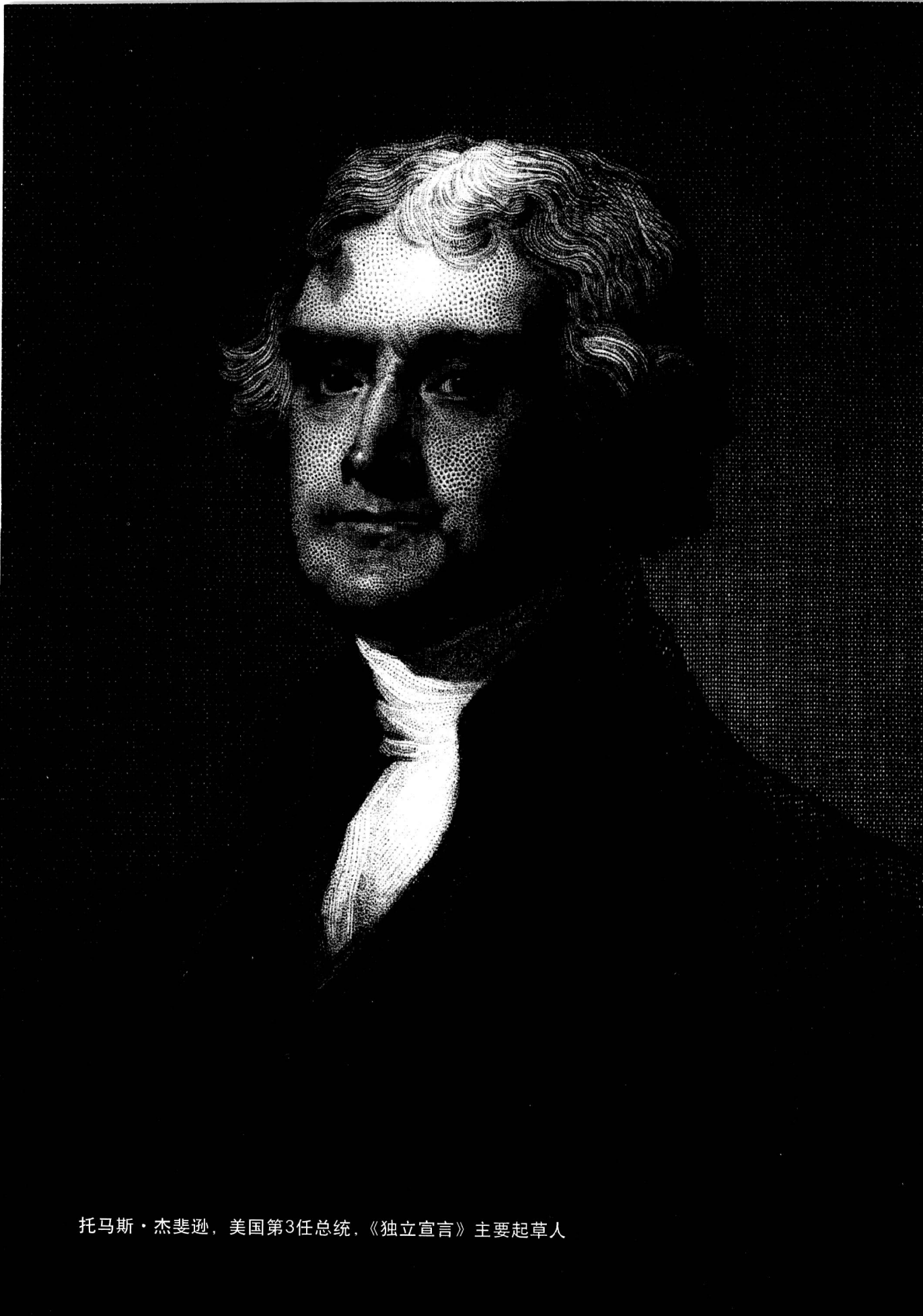
4. Svensson和Kristiansen（2002），“声学空间的计算机建模和仿真”。

5. Meehan （2002），“带压力的虚拟环境中的生理测定”。

6. Raymond (2001)，《The Cathedral and the Bazaar》，第4章，“神奇的大熔炉”。

第五部分

卓越的设计师



托马斯·杰斐逊，美国第3任总统，《独立宣言》主要起草人

第19章

卓越的设计来自卓越的设计师

而非来自卓越的设计过程

SEI（软件工程研究所）致力于软件过程成熟度方面的工作，其背后的基本假定就是软件产品的品质在很大程度上取决于它所使用的软件开发和维护过程。

——Mark Paulk (1995),《The Evolution of the SEI's Capability Maturity Model for Software》

……虽然某些人可能将他们视为疯子，但我们将他们视为天才，因为那些疯狂地认为他们能改变这个世界的人，确实就是改变这个世界的人。

——史蒂夫·乔布斯,《Apple Commercial》(1997)

卓越的设计和产品过程

开篇两段引用的作者的分歧大得不能再大了。谁是对的呢？

我以前曾列出了一些知名计算机产品的个人分类清单，它是根据产品是否拥有充满激情的粉丝来划分。这张清单扩展如图19-1所示，增加了一些最新的产品。我相信这种划分体现了设计之中的某些卓越之处。（这并不完全对应于商业成功，商业成功除了设计品质之外，还取决于许多复杂的因素。）

关于这张表有一个惊人的事实，根据我的判断，右手边的每件产品都是由一个正式的生产过程产生的，这种过程有许多的输入信息和批准过程。左手边的每件产品都是正式生产过程之外的产物。

其他领域的例子还包括：原子弹、核潜艇、弹道导弹、隐形飞、喷火（Spitfire）战斗机、青霉素、间歇雨刷。这些发明都是由一个小团队完成的，自然或有意地避免了正式的产品过程。

是	否
iPhone	Cell phone
Apple II	PC
Macintosh界面	Windows
UNIX	z/OS (MVS)
Pascal	Algol
Fortran	Cobol
Python	Appletalk

图19-1　计算机产品粉丝俱乐部

（仿照《人月神话》中的图16-1）

产品过程：优点和不足

这种观点，即使是经常正确而非一贯正确[1]，它向我们提出了以下重要的问题：

- 为什么这么多卓越的产品是在产品过程之外产生的？
- 产品过程的目的是什么？为什么要有产品过程？
- 人们能在产品过程中产生卓越的设计吗？如何做到？
- 我们如何让产品过程能够鼓励并促进卓越的设计，而不是产生阻碍？

产品过程抑制了卓越的设计吗

我相信标准的公司产品设计过程确实妨碍真正卓越和创新的设计。考虑到公司过程演进的方式和原因，这一点不难理解。产品过程的存在是为了给开发新产品的自然混乱带来秩序。

从过程的本质上来说，它是“保守”的，目的是将类似而又有一些不同的事情纳入到一个有序的框架中。因此实际上不同的、高度创新的设计就不适合这个框架。请考虑个人计算机，与20世纪60年代的企业级大主机或70年代的中心部门级小型计算机相比，它完全不是一回事情。

从过程的本质上来说，产品过程的目标具有可预测性：产品大致上根据业务需求来确定，然后再由卓越的设计师在这个问题上花相当多的时间，目的是按指定的时间和指定的价格交付产品。可预测性和卓越的设计并不是相处融洽的好朋友。

从过程的本质上来说，产品过程是“打响上一次战役”，它鼓励使用以前有效的战术，不鼓励使用以前失败的战术。因此对于面对新战役的产品（全新的需求或操作模式），这两类战术可能都不管用。请考虑iPhone，它和简单的移动电话根本不是同样的东西，和贝尔发明并由AT&T垄断的固话通信器材相比就差得更多了。

从过程的本质上来说，产品过程是“面向否决”的，目标是阻止不好的想法并且抓到疏忽。过程的目标是防止产品的销售不能达到预期，防止产品的成本超过预期的交付成本，防止承诺的功能和时间进度不能够实现。更微妙的是，公司的产品过程也是为了防止混淆产品线，以免自己的一件产品与另一件产品成为死对头，客户不知道该如何购买。因为失败可能由许多原因导致，产品过程通常要求许多人达成一致，每个人分别是一个可能的失败原因方面的专家。

这种一致性从几个方面抑制了卓越的设计。首先，每个专家拿钱是为了避免错误，而不是

为了创造卓越的产品，所以每个专家都偏向于找理由不前进。即使在真正的新产品没有被否决时，一致性机制通常会强制妥协，从而磨去棱角。但这些棱角正是最先进的！

其次，产品过程不只需要在现在的情况下达成一致，而且需要与过去达成一致，正如写到规则中的内容。产品过程会增长，它完全由各种规则组成，每次失败经验都带来新的规则或新的批准，以防止这样的失败再次发生。没有什么能阻止这些额外规则的产生，而它们一旦产生，没有力量能够消除它们，直到危机到来。按照事物发展的本质，官僚主义滋生，过程变得更重，组织机构随着成功和成长而变得不敏捷。[2]

20世纪60年代早期，当我在管理IBM的System/360计算机系列硬件开发时，我们的S/360 20型主机正在德国Böblingen的IBM实验室开发。这个团队拥有卓越的人才和强大的领导。但是，尽管他们已经运作了许多年，并为一些专门的市场生产了许多成功的产品，但是他们却从未成功地让一个产品进入IBM的主流产品线，即投入全球市场。对于“拿公司打赌”的System/360项目来说，这是很惊人的，所以我前往调查。

原因后来清楚了。这些工程师一直小心地、顾虑重重地遵守IBM公司的官方产品过程，按照超过100页的过程手册文档来执行。在其他实验室中，成功的项目经理在适当的时候大胆地让过程规则产生“例外”，成功的秘诀就在于聪明地选择！[3]我派送了一个有经验的过程操纵者去管理那个项目。他让那里的人才们意识到了自己的潜能。S/360 Model 20明显成功了。

最后，一致性的过程占用资源，却让创新的设计挨饿。一致性的构建过程要开会，开很多很多会。会议会花时间，花很多很多时间。卓越的设计师非常稀有，他们的时间极为宝贵。

为什么要有产品过程

我是在充满幻想地宣传革命性的颠覆所有公司的设计过程，偏爱创造性的混乱吗？不是的。前面提到的采用过程的许多理由都是不可避免的。有时候，必须得到公司的批准才能继续下去；有时候，产品进度计划和预算必须得到公司同意。这里的诀窍是把“过程”放在一个足够长的时间中，允许卓越的设计诞生，这样一来，当卓越的设计摆在桌面上时，要争论的问题就少了——不要把不成熟的东西拿到桌面上来讨论。

跟随式产品。产品设计过程总是会承担重要的作用，因为大多数设计的目的并不是要高度创新。这样做有充分的理由。当用户拿到了成功的、真正创新的产品时，至少有4种独立的效果会随之而来：

- 使用时暴露了一些缺点，需要在随后的产品中修正。
- 用户将这种创新用于意想不到的用途，扩大了产品的概念，通常以增量的方式进行。
- 创新所展示出来的优点促进了对更强大产品的需求，客户愿意为更强大的产品支付更多。
- 但在这些变化驱动的同时，流行也带来了锁定效应；用户不希望下一个产品是“革命性的”——他们想要熟悉并喜欢的产品。

因此，跟随式产品受到的限制更多，真正创新的空间更少。同时，成功为跟随式产品带来了更多的机会和可能的方式。但是没有哪个组织机构能做所有的事情。所以这些跟随式产品必

须在可能的范围内小心地选择。它们的开发必须受到监控，确保产品能够完成它选择的目标，并达到其目标客户的要求。

产品过程的重复循环是：

- 产品定义
- 市场预测
- 成本预估
- 价格预估

它有效地实现了这种选择和监控。

但是，准确的市场预测取决于对类似产品的一些销售经验。准确的成本预估取决于对类似产品的一些开发和制造经验。

因此，如果说产品过程是为跟随式产品设计的，那么这也是恰当的。对于创新而言，人们必须跳到过程之外。

提高设计实践的水平。产品设计和发布过程不能让优秀的设计师变成卓越的设计师。没有卓越的设计师，这些过程基本上不能产生卓越的设计。但是这些纪律能够提升较低水平的设计，改进平均的设计实践。对此无可厚非。

软件工程社群对开发过程一直给予关注。这是需要的，因为我知道少数设计社群的平均实践目前还落后于最佳实践，并且最差实践目前还落后于平均实践。

Watts Humphrey和软件工程研究所致力于能力成熟度模型（CMM）方面的工作，并且在推进其采用方面的努力很有价值。[4] CMM是对好的设计过程的各个方面的一组严格测量标准，这些方面常见且有效。如果一个设计公司进行CMM评审且得分很低，就不只要看它自己的实践，而是要看看那些成功的公司的实践。过程改进对于提升社群实践的最低水平是具有价值的。CMM在这方面做得非常好。

这里没有魔法。过程改进不能提升社群实践的最高水平。卓越的设计不是来自卓越的过程，它们来自努力工作的人才。人们引用苹果公司的名言："我们处于1级（在成熟度模型中），并将永远如此。"[5] 但是苹果公司的成果胜过千言万语。

过程对于创新设计难道不也是必要的吗？有人曾问我："S/360项目要协调多个国际实验室和多个市场的需求，肯定使用了大量的过程。你们如何使用过程，而又不被过程抑制？"

我们的核心设计团队很好地隔离在一般的IBM监视过程之外，得到了几个层次的大胆经理的强大支持。我们有特别的权力，可以从公司其他团队中招募人才。我们有足够多的钱。

让设计变成产品需要走过标准过程。高层下命令，说大家都知道这次赌博是革命性的，知道它需要在标准过程上做一些让步。我们的团队每周都与过程团队搏斗，努力争取在市场预测、成本预估和定价方面追求更多创新。但是，过程团队的人才非常有条理，他们的技能对我们的

成功来说也十分重要。

观点碰撞：过程抑制，过程不可避免，如何处理

卓越的设计来自卓越的设计师，去找到他们

作为一个不好动、五音不全的人，我认为很明显，不仅才能的分布是不均衡的（无论是音乐、棒球投手、舞蹈还是设计），而且某一方面才能的差异也是很大的。

即使在一个由类似学历和经历的人组成的团队中，才能的差异也很大。我的一些团队成员和学生是极有天赋的设计师——他们在我的记忆中闪着光辉。这类天才像宝石一样点缀着艺术的历史。

而且，没有两个人具有同样的才能（我想上帝这样安排是为了让我们每个人都有一些独特的地方，有别于其他人，能够以独特的方式提供服务）。因此，聪明的领导者在他领导的人和能得到的人周围画上责任盒子，而不是把人放到梦想中绝对理想的盒子里。

我们正确且基本的民主思想是法律面前人人平等，而且拥有平等的机会。但追求这些目标不应该让我们无视才能分布的不均衡性，以及培养、打磨和表达才能的条件的不均衡性。

因此，我们的结构和过程必须无情地意识到，如果信任他们，那么给他们授权和自由，曾经完成过卓越设计的人更有可能完成另一项卓越设计。[6]

卓越的设计需要大胆的领导者，他们要求创新

首要的一点，组织机构的最高领导者必须对卓越设计的创新产品充满激情。在Steve Jobs领导苹果公司的第一阶段，情况显然是这样的，他的继任者让情况有所变化，当他回来重新执掌苹果公司时，情况又成为这样了。在Thomas J. Watson (CEO 1914～1956)和Thomas J. Watson, Jr. (CEO 1956～1971) 领导下的IBM的情况也是这样的。其他例子还有许多。

如何设计一个鼓励卓越设计的过程

我曾饱受沉重的过程之苦，我也有一些经验来绕过或违反这样的过程，但我没有剪裁或重组过程的经验。每个组织机构都需要不时地这样做。我希望将这项工作委托给一流的人才，让他们有巨大的权力，在有限的时间内完成任务。

如果你正在设计一个新产品过程，或正在重组一个原有的过程，应该怎样顶住压力，克服自然的阻力倾向呢？怎样设计一个过程，允许、支持甚至鼓励卓越的设计呢？

首先，产品过程必须明确基本重点以及对这些重点的约束条件，而且只确定这些重点。这在本质上是一种防范机制。它必须安全地保护皇冠上的明珠，但同样重要的是，它必须避免建造围绕垃圾桶的高篱笆。这需要判断力和限制——保护者的本能就是过度保护。第27章展示了在一个特定的案例中，分离识别具有基本重要性的事情是如何生效的。

其次，产品过程必须提供容易、轻便的例外机制，在项目经理请求时就能执行，只要高

一级的老板同意就足够了。换言之，判断和常识必须明确地提出并运用："所有规则都可以打破。"

寻求概念完整性：信任一名主设计师来完成设计

既然概念完整性是卓越设计最重要的属性，而且这出自一个或几个齐心协力的人，聪明的经理就会大胆地信任有天赋的主设计师来完成每项设计任务。[7]

信任意味着很多东西。首先，经理本身一定不能对设计放马后炮。这是真实的诱惑，因为经理很倾向于成为一名设计师，但经理的设计天赋可能不如他手下的人那么卓越（设计和管理是非常不同的工作），而且经理的注意力肯定要分散在其他任务上。

其次，必须十分清楚主设计师对设计拥有完全的权威，虽然他可能只管理几名助手，但是他在职位上与项目经理是平等的。

再次，主设计师必须不受项目以外的观察者的影响，并且要防止分散精力。

最后，他必须得到他想要的工具和帮助。他要做的事情是最重要的。

注释

1. Air Force Studies Board(2008)，《Pre-Milestone A and Early-Phase Systems Engineering》，表明在许多最成功的、最具创新的武器开发项目中，这一点都是正确的。

2. 这一普遍的现象也存在于软件中，Lehman和Belady的经典研究令人信服地记录了这一点，他们研究了在IBM运营System/360的过程中熵的增加：Lehman and Belady (1971), "Programming system dynamics." 更全面的论述参见一篇重要的论文：Lehman and Belady (1976), "A model of large program development"。

3. 参见文章 "Template zombies," DeMarco (2008)，选自《Adrenaline Junkies and Template Zombies》的第86章。我不相信这个IBM实验室遭受了模板僵尸的"形式与本质的对决"病的折磨，而只是谨慎地遵守远处寄来的编写规则。

4. 这一点得到了承认，并由此获得了2005年的国家技术勋章。

5. Atlantic Systems Guild公司的James Robertson在2008年告诉我这句苹果公司的名言（私人交流）。

6. Cross (1996b), "Winning by design," 是关于Gordon Murray方法的有趣的案例分析，Gordon Murray是获胜的赛车设计师。这篇报道介绍了一个主设计师团队，避免了大多数类型的过程，并从约束条件中受益。

7. 英国副首相要求皇家工程学院（Royal Academy of Engineering）提供一份关于如何让雨天行车更安全的报告。RAE将这个任务代理给了只有一个人的委员会：它的主席David Davies爵士。这份报告以破纪录的时间完成了，干脆而坦率，充满了具体的建议（Davies (2000)，《Automatic Train Protection for the Railway Network in Britain》）。

读书笔记

John Cocke (1925～2002), 计算机天才

第20章

卓越的设计师从哪里来

天才在没有训练的情况能够生存并取得成功，但这不会减少对浇灌和修剪的回报。

——Margaret Fuller (1820—1850)，摘自个人日记

每个超出一般水平的人都接受了两种教育：第一种来自他的老师；第二种更私人也更重要，来自他自己。

——Edward Gibbon (1789)，《Memoirs of My Life and Writings》

我前面刚论述了卓越的设计来自卓越的设计师，而不是卓越的设计过程。尽管技术设计现在总是团队的工作，我们仍然能识别出一些卓越的设计师，团队围绕他们而组建：John Roebling（钢索布鲁克林大桥）、George Goethals（巴拿马运河）、R. J. Mitchell（“喷火”战斗机）、Seymour Cray（CDC 6600，Cray I型超级计算机）、Ken Thompson和Dennis Ritchie (UNIX)。

对于高科技设计，特别要关注的是单人设计模式和团队设计模式之间内在的冲突。单人设计模式在艺术、文学和工程领域产生了卓越的作品，而团队设计模式则是现在复杂的制品和经济的节奏所要求的。

- 如何才能培养卓越的设计师？
- 如何形成一些设计过程，支持并增强卓越的设计师，而不是限制他们，从而使作品同质化呢？
- 如何让团队为卓越的设计师提供最好的支持？

我们必须教他们设计

我们对设计师的正式教育通常是完全错误的。Schön[1] 说过类似的话：技术理性认为所有的专业人士都应该像现在教育工程师那样去教育，即先是相关的基础科学和应用科学，然后是应用的技能。

熊恩在这一点上强烈反对技术理性。他辩称，所有专业技能都是靠“批评性实践”来掌握

的。他认为在医药、法律、政府、建筑、艺术、音乐、社会工作等领域中都是这样的，实际上在工程领域也是这样的。医药教育意识到了这一点，从第3年起，学生就要花越来越多的时间在诊所、大查房中，对病人负责。建筑教育从未错失这一真理，所以多年来一直是工作室主导了整个行业。

另一方面，在美国，绝大多数工程设计师将大多数正式教育的时间花在了教室里，或者在实验室里做规定的实验，不去做准备接受批评的设计。软件工程师尤其是这样。

我认为Schön的观点是正确的。在这样的教育中，我们错误地使用了教育过程中最宝贵的东西：学生的时间。工程学校正逐渐将批评性设计实践重新加入到课程设计中，尽管这需要很高的师资投入。

有人认为，硬件或软件工程师必须知道实践背后的基础科学知道，而且应该先知道。最现代的工程教育驳斥了这一点，在大学一年级就开始了批评性设计，与科学教育同时进行。计算机科学课程很少这样做。

类似地，好的工程课程通常包含“合作”或“三明治”计划，学生通过这些计划，在初始的学院教育和最后的学院教育之间穿插工作岗位实践（以及公司培训）。这种方式在计算机科学课程中仍然很少。

太过学院派的正式教育，其弱点在于它依赖讲课和阅读，而不是批评性实践。关于设计风格的有效教育会让学生用Cray的风格设计一个有约束条件的计算机架构，用巴赫的风格设计一首赋格曲，或以Wren的风格设计一幢建筑。一名知识丰富而且有眼光的指导者将指出风格上不一致的地方，并针对约束条件，对设计的整体优点给出批评意见。

在工程和计算机科学领域中，这样的批评意见需要指导者具有一定的信心，甚至是胆量。我们对科学的强调可能让我们不愿意卷入形式自由的主观批评，对提出主观批评也没有什么经验。但这在教学设计中是很重要的。

学生之间也可以通过这样的批评实践进行相互指导。设计师学习其他设计风格的最佳方式，就是承担起向其他设计师教授这些设计风格。

我们必须为卓越设计而招募人才

经理常常招聘一些新的设计师，然后下意识地以经理自己的工作为标准对他们进行评判：“他能做好我现在做的工作吗？”这样会倾向于能说会道的领导者以及善于开会的人。同时会忽略内向的、说话慢的，尤其是不合常规的人。但优秀的设计师也可能就来自这样一些人！（我并不是说优秀的设计师更有可能来自这类人，我不知道。）我们的经理忽略了这些天才，这是我们的损失，也是这些天才和社会的损失。

我们怎样才能更好地选择？首先，要提醒我们自己要找怎样的人。其次，查看设计工作本身的组合方式，而不只是工作的口头介绍。例如，微软就有候选手艺计划，这在软件工程公司

中还不太常见。

我们必须深思熟虑地培养他们

大多数重要的工业组织和军方组织都已具备精细的、充分发展的过程，将工人培养到经理，再到执行官，直至最高执行官。在每个职业阶段，针对新的中尉、少校和将军，还有另外的教育计划。有前途的人才很早就被发现并进行追踪。组织为他们指派指导者。轮换任命让他们具有经过仔细计划的多样经验。最有前途的人才被安排在快速通道，作为顶尖专业人士的助手。最有前途的年轻律师被招聘为最高法院的职员。

上帝用这种方式培训了许多领导者。通过一次“意外”，摩西接受的培训是领导一个民族，他在法老的宫廷里作为王子长大。大卫作为扫罗王的竖琴师，看到了一个王国如何运作，如何做出正确的判断。

在大多数技术型组织机构中，我都没看到他们考虑打造类似的、现实的方式来培养非管理型的技术领导者，对于公司所依赖的卓越设计师来说，这样的方式就更少见了。

让两架梯子真实而体面

培养设计师与培养经理不同，第一件事就是要为他们打造一条职业发展道路，让他们的报酬和社会地位能够反映他们对创新企业的价值。这通常被称为“两架梯子”。我曾在别处讨论过这个问题，这里我只想重复一点，为对应的职位支付对应的工资是容易做到的（市场的力量通常会造成这种效果），但需要很强的主动措施才能给他们同样的威望：同样的办公室，同样的人员支持，在职责改变时提高反向偏压[2]。

为什么需要特别注意“两架梯子”？也许因为管理者也是人，从内心深处容易认为他们自己的任务比设计更难、更重要，需要小心评估如何发扬创造性和创新性。

规划正式的教育经历

新秀设计师和新秀经理一样，需要持续不断的正式教育，以及穿插其间的真实而精彩的实践，也需要一位高水平的设计师提供批评性指导。

为什么需要正式教育？今天的世界在不断地改变。在高科技学科中，快速更新的技术教育是显而易见的，几乎让人恐惧。自从1952年我进入计算机领域以来，我的职业智慧人生一直像在大海中与拍岸的大浪搏斗。在刚刚被一个浪推倒之后，下一个浪又来了！令人激动、令人喜爱，永远在变化。所以正式教育的第一个理由就是不断再培训。

我发现，正式的短期课程是再培训的有效且经济的方法，我每年都参加一次。为什么？人们不能够通过阅读杂志和参加会议来赶上时代吗？人们确实可以！但我非常相信正式教育：一名好老师精心准备好一个主题的全面综述，这可以让我的学习效率提高两个以上的数量级，很快掌握全面的观点。否则，这些观点可能需要研究几十本杂志才能获得。作为一名教师，我向我的客户提供这种学习效率，同时我也为自己购买。

第二个理由是加深和加宽。实现这一目的最有效的方式就是同时学习以前和当代的优秀设计和糟糕设计。出于这个目的，正式教育提供的首要好处就是分小组——专业教师更易于学习竞争的设计概念和风格。正如设计公司内部文化会强调它自己的传统和观点，公司资助的正式教育也是如此。这是新秀设计师和他们的指导者向外部寻求正式教育的最好理由。

作为一名IBM经理，我做的最有成果的一件事情与产品开发没有关系。这件事就是将一名有前途的工程师作为IBM的全职雇员送去密歇根大学读博士学位。这件事在那时对一名繁忙的计算机架构经理来说似乎只是一件非常偶然的个人决定，但对IBM的回报超出了我的想象。Ted Codd的博士学位为他的研究生涯奠定了基础，他的研究导致他发明了关系数据库的概念并获得了图灵奖[3]。在25年的时间中，关系数据库是IBM利润最好的计算机产品线上的首要应用程序。

规划不同的工作经历

就像最好的组织机构为新秀经理所做的那样，我们也需要为新秀设计师做那些事情。这里的关键词是“计划”，年轻人的课程本身也需要设计：针对多样性、针对参与的深度、针对盘旋上升的挑战和责任。

通常最有效果的早期工作指派是让年轻的设计师在他们设计的产品的用户组织中服务。我自己的一些短期工作为我深入理解计算机用户的需求提供了巨大的帮助：我曾在一家商业数据处理公司编写了40个州使用的工资表程序，在一家科学计算公司计算火箭弹道，在一家密码公司工作，在一个电话交换实验室确定4方通话的拨号者，在一个工程物理实验室测量微小的大地振动。正如我在前面所说的，我曾做了两周的计算机操作员学徒，在一间玻璃房里挂磁带，这在我设计操作员控制台时给我带来了真实的冲击。Dewey说我们在操作中学习，他是对的。对设计师有效的成长课程必须包含各种经验。

规划离开组织机构去休假

处在职业发展中的设计师可以通过离开组织机构而恢复精神、开阔视野——也许受聘于客户，也许在大学教书，也许接受联邦政府机构的任命。防止创造性人才停滞不前是一项很好的投资。

管理他们时必须发挥想像力

John Cocke-Ralph Gomory的故事。在我认识的人当中，John Cocke可能是最卓越的人，而且肯定是最有创造性的人。我们同时在1956年7月加入IBM，参加了Stretch超级计算机项目，那时我们刚刚拿到博士学位，我们先是在一个大房间，后来共用一个两人办公室。这种安排很好，因为John一个人在晚上工作，我在白天工作。John对计算机的所有方面都充满激情。我怀疑他一天里思考计算机的时间超过思考其他事情的时间之和。他不仅深刻理解计算机，而且理解所有支持学科和技术[4]。他是一个少见的组合体——既深入思考，又外向开朗。就像《Ancient Mariner》中所说的，“他从三个行人中拦住一个”，解释他最新的思想。没法不喜欢

他——亲切，大方到极点，总是很开心。Harwood Kolsky的回忆录生动地记录了John的个性、风格和态度[5]。

Cock吸引卓越的协作者，这些协作者帮助记录和实现他多得难以置信的想法。他的想法中有三个都值得获得图灵奖，分别是与Kolsky协作的指令管道（instruction pipelining）[6]，与Fran Allen and Jack Schwartz合作的全局编译器优化[7]，以及与George Radin合作的精简指令集计算机架构[8]。

那么像John Cocke这样一个特殊的天才，不能管理一个小组，几乎不发表任何东西，怎么能够作出这么多重要贡献呢？其实这个故事中有两个天才，另一个便是Ralph Gomory——IBM研究主管和科学技术高级副总裁。像Cocke一样，Gemory也因他自己的贡献获得了国家科学奖章。

Gomory创造了一个组织、一种气氛和一种组织管理风格，目的是让IBM研究部门的每个人都能够以最适合自己特殊才能的方式作出贡献。Ralph说："我对John和对其他人是一样的。"但他的话没有讲出要点：他对每一个杰出下属都是不一样的——完全根据他们的天性和需要。他也曾很自豪地说："John是IBM研究部门里薪水最高的人，因为他的贡献最大。"[9]

必须严密地保护他们

防止他们分心

如果我们有了卓越的设计师，就希望他们进行设计。设计效率要求"流畅"（flow），即一种不受打扰的、高度创造性的、高度集中的精神状态。我们这些设计师都体验过，并希望得到这种快乐。在《Peopleware》一书中，DeMarco和Lister对流畅及其重要性，以及如何实现流畅进行了极好地讨论[10]。

现代组织机构中存在许多障碍和分心，阻碍流畅的发生：

- 会议
- 电话
- 电子邮件
- 规则和约束条件
- 职员官僚主义和"服务"小组，他们制定规则只是为了简化自己的工作
- 客户
- 专业拜访者和记者

许多创造性组织采用了"安静的早晨"这样的过程来强化流畅。当苹果引入第一台个人计算机，IBM试图赶上时，CEO John Opel在Boca Raton建立了一个实验室（现已关闭），用于开发IBM的项目。其他IBM的员工，甚至包括有相关职责的协作员工都不允许进入。

类似地，我对非项目拜访者关闭了System/360项目，包括IBM员工和客户，时间从1964年

2月到4月。我们有太多的工作需要流畅[11]。

Jeffrey Jupp是英国空客的技术主管，我对他进行了访谈，讨论空客的机翼在英国设计和制造，而机身却在法国设计和制造。后来，在我们的对话中我问他是否能够与他的主设计师谈谈，他回答说“不行”。我理解并尊重这一点。

保护他们不受管理者干扰

平庸的、不安全的经理可能会扼杀设计师的创造性。平庸的经理常常不能意识到团队中的宝石。有时候他不能认识设计对于团队成功的重要性。有时候他不能理解他自己为设计魔法提供支持的职责。

有时候经理怨恨或不愿承认“下属”设计师是更好的设计师。当优秀的设计师拿的薪水更多时，有时候经理会觉得受到冒犯。结果是缺少鼓励、缺少帮助以及恶意的贬低。

于是高级管理层的任务就清楚了：他们必须积极改变一线经理，最好是提升他们对自己才能和特殊角色的看法，在团队鼓励和领导力方面提供培训。

防止他们去做管理

我曾看到过一些潜在的卓越设计师从设计领域转到管理领域。他们从未充分发挥潜能。然而，我们组织机构的文化却鼓励甚至强制这样做。反抗这样的企业文化需要意愿、拒绝和决心。

Seymour Cray就是一个有启发意义的例子，他是有史以来最卓越的超级计算机设计师之一。Cray是第一代真空管计算机的设计师，地点是在明尼苏达的St. Paul，先是在工程研究协会(Engineering Research Associates)，然后是在控制数据公司（Control Data Corporation)。为了设计CDC 6600，他将“包括看门人共35人”的团队隔离起来，并让自己从所有其他CDC事务中解脱出来[12]。

当6600的巨大成功让他再次陷入CDC的管理时，他选择了离开，在CDC的祝福下，在一个僻静的牧场建立了Cray计算机公司。他个人监督了Cray 1型计算机的所有方面，从线路到冷却，到Fortran编译器。

然后，当Cray计算机公司因成功而壮大，再次让他陷入管理时，他将一个团队带到科罗拉多，建立了Cray研究公司。直到一名醉酒的驾驶员结束了这种有决心的模式[13]。

把自己培养成一名设计师

假定你是一名技术设计师，并希望提高。有没有来自你的学科之外的建议能够有所帮助？我认为是有的。你现在必须开始规划自己的成长道路[14]。这事只有你自己负责。

不断画设计草稿

设计师通过设计来学习设计。有些草图需要很详细，因为魔鬼确实隐藏在细节之中，许多

宏大的规划是建立在埋没的石头之上的。达·芬奇的《Notebooks》就是一个很好的例子，它说明了这种最佳实践。有志向的年轻软件设计师也可以保存一个笔记本，记录下自己遇到的模式和设计时的发明。

寻求有知识的人对你的设计进行批评

Donald Schön在他的大作《Educating the Reflective Practitioner》中，认为大量的批评性实践实际上是唯一成功的教学方法。他引用了一个又一个的学科（法律、医药、建筑、土木以及中世纪的工匠行会），说明他们都已进化成了这种教学方法（可能是独立的）[15]。现代的博士论文就是用这种方法来教授科研实践的。

研究教学示例和先例

在这项实践中，你可以模仿许多卓越的设计师。Robert Adam研究过Christopher Wren。Wren研究过Palladio。Palladio请求他父亲支持他去罗马，对优秀的罗马建筑进行测量和写生。罗马人研究并融合了伊特鲁里亚人（Etruscan）和希腊人的建筑风格。每个卓越的设计师都掌握了前辈的丰富遗产，然后加上他自己的新概念。

正确研究教学示例要求有虚心的态度。这些先例的名望经受了几个世纪的批评，必然有其优秀之处。在较新的领域里，能研究的范围保底只有几十年。但不论先例的池子有多深，学生的任务就是寻找并掌握这些先例的杰出之处，即使他的沉思或新的环境让他随后转向完全不同的方向。

计算机架构师需要学习各种商业制造的机器。有些人认为它们足够好，值得投资的那些钱。（还有很多发表的架构，只是这些架构没有这样严格测试，因此不值得深入研究。）

在研究设计先例时，关键要假定它的能力——正确的问题是“什么导致了一位聪明的设计这样做？”

而不是

“为什么他干了这样一件蠢事？”

通常，答案就藏在设计师的目标和约束条件中，发现这些目标和约束条件常常会带来新的深刻见解。在夫妻间的分歧中，对于“你为什么...？”或“你为什么不...？”的聪明答案是“没有感觉”，这通常也是真正的答案。但在研究设计决定时，这很少是真正的答案。

如果可能，听听同时代的设计师们如何讨论他们的工作。如果有可能，阅读设计师们针对他们的工作写下的文字。

Gerry Blaauw和我发现这样做很有益，即将我们对其他计算机架构的研究投射为一种通用格式（通用结构、标准草图比例、通用散文描述元素、通用正规描述语言），再加上简单的文字评价每种架构的亮点和特性[16]。

一个自我教育项目：1 000平方英尺房屋的建筑平面图

从北卡罗来纳州立大学设计学院的设计入门课程中我们可以得到了针对设计师的一个有用的自我教育练习，这与设计师的设计学科是什么无关。

项目。设计1 000平方英尺家庭住房的建筑平面图，这个家庭包括父母和两个小孩，3岁的儿子和6岁的女儿。地点在北弗吉尼亚，郊区，离街道50英尺，径深70英尺，有一些树木，朝南。

日记。用标明日期的日记记录下你的设计问题、决定和理由。下面是要考虑的一些问题：

- 虚构一个更详细的建筑项目，利用你的想像力，写进你的日记。
- 从给定的项目中你可以导出什么约束条件？
- 哪些是列入预算的物品？你如何管理？
- 你打算满足哪些必要条件，显式或隐式的？
- 你如何判断两种设计哪一种更好？
- 你是否使用CAD工具？如果用了，评价一下在任务的不同阶段使用CAD工具和草图的优缺点。
- 你如何推进？分析你的日记并勾画出你的设计轨迹。
- 评估：你的设计中有哪些优点？有哪些不足？

注释

1. Schön (1986),《Educating the Reflective Practitioner》。

2. Brooks (1997),《人月神话》，118~120, 242。

3. Edgar Frank Codd, 如果你想寻找他的著作，请用这个名字。

4. 甚至在他最后生病期间都离不开椅子，Cocke仍然让我享受到了一些新科学和他最新的想法，以及如何应用于计算机。

5. http://www.cs.clemson.edu/~mark/kolsky_cocke.html, 我在2008年11月26日访问了这一网址的内容。

6. Cocke and Kolsky (1959), "The Virtual Memory in the STRETCH Computer."。这实际上是在讲指令管道，而不是我们今天所知的虚拟内存。

7. Cocke和Schwartz (1970),《Programming Languages and Their Compilers》；Allen和Cocke (1971), "A catalog of optimizing transformations"。

8. 精简指令集计算机（RISC）的概念常常被误解。其基本思想不是指令的精简集合，而是精简指令的集合，也就是说，更简单的指令。在极端的形式下，没有派生的指令，甚至没有移N位或乘法指令。这使得累加器-加法器-累加器循环最小化，利用指令缓存和一个优化的编译器，所有任务都执行得更快。我知道除了John这样既掌握计算机设计又掌握编译器优化的人之外，没人能够像这样结合这些概念。George Radin 是重要的协作者，但最初的论文Radin (1982), "The 801 mini-computer," 和Radin (1983), "The IBM 801 minicomputer," 应该加上

Cocke的名字，作为第一作者，虽然我猜想他一个字都没写。

9. 与Ralph Gomory的私人交流（2008年11月）。

10. DeMarco (1987)，《Peopleware》。

11. 在本书的网站上，有我在1964年2月4日发给市场部门领导、我的老板和实验室经理的信的节选。

12. Murray (1971)，《The Supermen》。

13. http://americanhistory.si.edu/collections/comphist/cray.htm，我在2009年8月12日访问了这一网址的内容。

14. 我看到的关于规划学术生涯的最好建议是Gilbert Highet在《Art of Teaching》(1950)给出的，具体见其第21页：

当一名年轻的德国学者开始他的职业生涯时，他通常会选择3、4个真正感兴趣的领域，这些领域有大量的工作需要去做，而且重要的是这些领域都相互关联，而且最重要的是他觉得这些领域汇聚在他的主题中心上。他会对这些领域写几组课程，加强并丰富每组课程，直到形成一本书。如果他的精力和理解力足够，他会因此成为3、4本书的作者，每本书都推荐并阐明其他的书。然后他会继续……年复一年地、战略性地扩大（每个领域），直到他积累起整个主题的真正权威的知识。……以这种方式规划他们的学习和教学的学者，通常发现……他们有足够的兴趣和差不多的知识来担任3个职业。

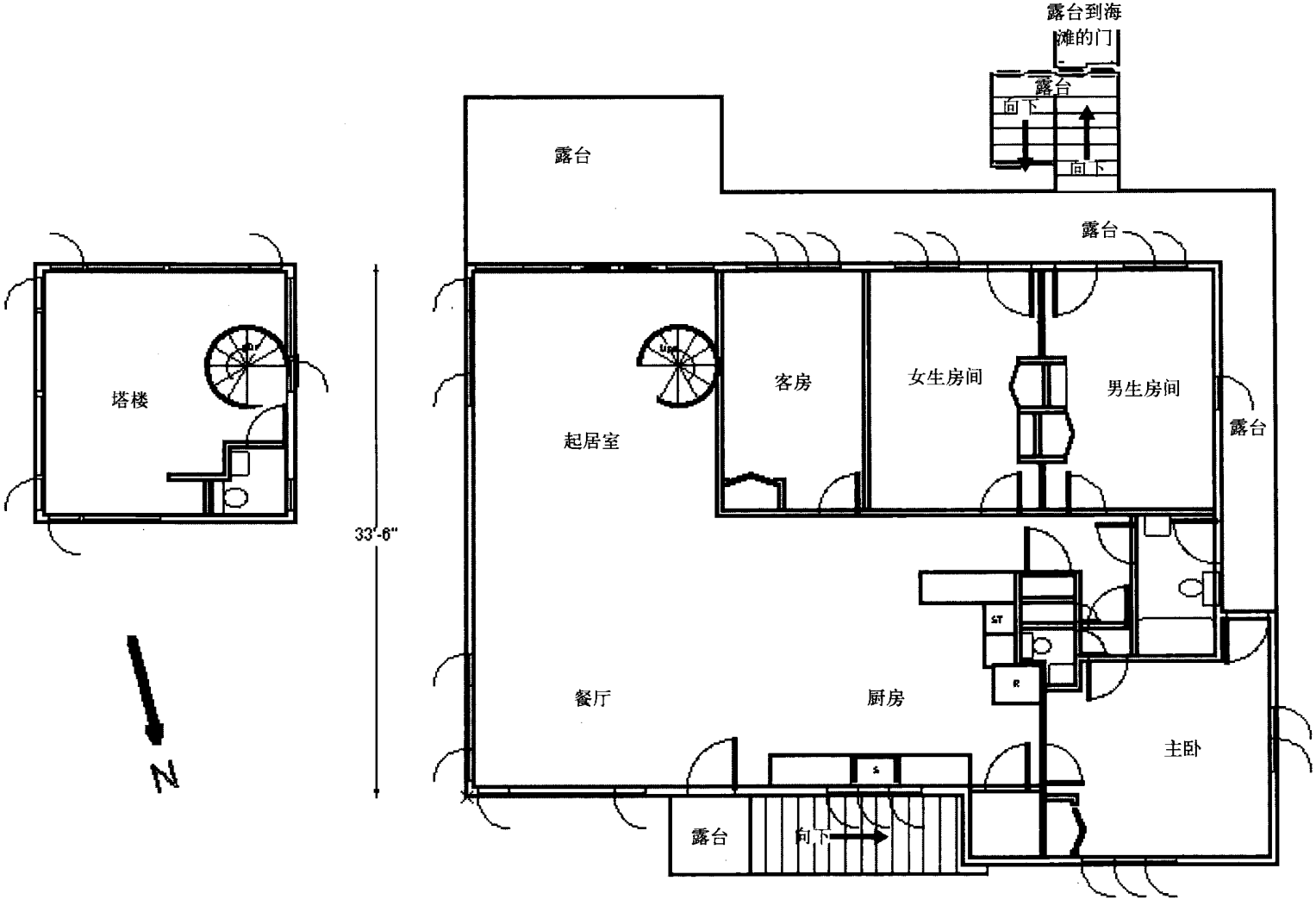
15. Schön (1986)，《Educating the Reflective Practitioner》。

16. Blaauw和Brooks (1997)，《Computer Architecture》第9～16章，“A computer zoo.”标准格式在第9章中介绍。

第六部分

设计空间之旅：案例研究

回顾一下，大多数的案例研究都有一个突出的共性：最大胆的设计决定，不论是谁做出的，都对好的结果产生了重要影响。这些大胆的决定有时候是因为愿景，有时候是因为铤而走险。它们一直都是赌博，要求更多的投入，希望得到好得多的结果。



“View/360”海滨小屋的主层和塔楼平面图

第21章

案例研究：海滨小屋“View/360”

世界上最美丽的房屋（就是你自己建造的房屋）。

——Witold Rybczynski (1989)

亮点和特性

为什么是这个例子？它记录了一个简单的、可以理解的结构，说明了如何必须做出许多决定，以及影响这些决定的无数考虑。

大胆的决定。让小屋尽可能靠近大海，同时又仍然在有担保契据的地块上。它比所有邻居的小屋大约向前延伸了40英尺，被潮水冲走的风险更大一些。

需要精打细算的资源。对于这座小屋的设计来说，后来发现需要精打细算的资源是朝海一面的长度，亦即景观和微风。

旋转楼梯的巧事。木楼梯因底楼空间压缩而添加，后来发现它是一件特殊的艺术品，是视觉上的亮点。

构建期间的改动。在构建期间所做的设计改动极大地改进了视觉感受、对小屋的感觉和它的价值。没有利用构建期间所有的改动机会，这是一个错误。

打桩的位置。业余建筑师和专业建筑师都没有仔细考虑如何将桩打在小屋的重心之下，没有仔细考虑桩的分布，以便让每根桩的承重大致相同。桩不均匀地打在沙土中，在应该有桩而没有桩的地方，小屋发生了下陷。

背景介绍

位置：

北卡罗来纳州，Caswell海滩，Caswell海滩大道321号，北纬33°53.6'，西经78°2.1'。该处是一个东西走向的岛屿，有一条主干道。一排地块位于大西洋和主干道之间，另一排地块介于

大西洋、恐怖角河（Cape Fear River）和它的湿地之间。小屋在朝向大海的一边，南偏西15°。

所有者：

Frederick和Nancy Brooks一家

设计师：

Frederick和Nancy Brooks，建筑设计师；Arthur Cogswell，美国建筑师协会会员，负责结构工程和塔楼屋顶

日期：

1972，框架和外墙完成并入住

1997，建造完成

1972年8月当地家庭成员：

父母：Frederick和Nancy

孩子：Kenneth，14岁；Roger，10岁；Barbara，7岁

奶奶：Octavia，71岁

亲密的小孩朋友：Chandler，10岁

目标

首要目标。我们的首要目标是为家庭成员和朋友建造一个舒适的、非正式的度假小屋，充分利用朝向大海的自然景观。这个小屋不打算出租。

其他目标。

- 充分利用景观。
- 设计随意的、不张扬的、休闲的内部装修。
- 充分利用白天和晚上的海风。
- 睡下14个人，供22个人同时就餐。
- 提供一间奶奶与客人房、一间主卧、一间男孩卧室、一间女孩卧室，共4间卧室。
- 提供足够的淋浴和卫生设备。
- 建造的小屋应该能承受风速大于100英里/小时的飓风。飓风的危险每年会出现两次，10年会有一次真正的袭击。
- 设计一间一人用的厨房，但也可以容纳4～6个人同时工作。
- 隔离男孩和他们的朋友的噪声。
- 保持较低的维护要求。
- 让这间小屋成为大家一起参与的项目，让家庭成员得到锻炼，并增进彼此间的感情。

机会

建筑位置。这块地朝向大海的一边有75英尺。东南面是秃顶岛（恐怖角）和船只进出恐怖角河的漂亮景色。“前面”定义为小屋朝向大海的一边，而不是朝着道路的一边。地面是粗糙的海沙，植物是低矮的灌木、海燕麦和菠萝属蔓藤。

沙丘。这间小屋可以离最高水位标识仅65英尺，因为它受到一排沙丘的保护。

景观。因为这个岛屿是狭窄的，所以小屋不仅在前面有180°的海滩景观，还有背面的135°的恐怖角河及其湿地的景观。

微风。小屋的位置自然朝向南偏西15°。主要的海风是南风和西南风，在温暖天气的多数时间里都有。

约束条件

预算。没有足够的钱一次建造完整的带4间卧室的小屋。

时间。在任何一个夏天，家庭用于建造小屋的时间都是有限的。

法规与契据要求。

- 小屋必须建在16英尺的桩上，其中8英尺必须在地下。
- 地块每边要求留出的距离是10英尺。
- 小屋必须是独户住宅。
- 与电力和化粪池有关的法规必须遵守。

前面的沙丘。前面的沙丘只能稍微处理，例如，在上面架设木板铺成的小道。

公共设施。这个位置只提供电力和水，没有燃气或下水道。

契据。自1938年绘制地图以来，这个地块朝向大海的一面大约增长了65英尺。我们对这部分土地有产权转让契据，没有担保契据。

外观。外观没有限制，我们也不认为这很重要。

设计决定

在一段宽松的时间里建造小屋。

- 马上建立起框架和外墙，里面是干燥的，可以宿营，一间浴室装好水管，安装好化粪池，为小屋提供临时电力。
- 将所有最初能提供的现金用于使建筑面积和窗户最大化。
- 家庭成员将完成所有内部工作，包括内墙、门、橱柜、电线，以及大部分的管道。

利用75英尺的地块宽度。大多数海滨地块的宽度是50英尺。大多数朝向大海的小屋是长

而狭窄的。为了充分利用55英尺的允许宽度，我们让小屋斜向一边，这样它的宽度就大于它的深度。因此需要一张定制的平面图，而不是书上的例子。

利用景观。

- 让小屋建在地块中尽可能靠前的位置，但仍在有担保契据的地块上。
- 在二楼前面建造一间视角最大的塔楼间，四面都是玻璃。

推论：屋顶沥青是受到限制的。屋脊不能够高于塔楼的窗台高度。

- 在所有高处装上大量的大型窗户。

推论：结构必须加强，以防歪斜。

利用微风。

- 让每间卧室都有朝向大海的一面。
- 计划让小屋对微风开放——不安装中央空调。

推论：预期到处都会有潮湿和盐沫。

- 在起居室的前面设两扇6英尺的推拉门。
- 提供足够大的前露台，带防晒设施。
- 使用带窗扉的窗户，可以最大程度打开，并调整方向，收集微风。

推论：窗户安装的方向朝向西南方或东北方。

建筑防潮。小屋既需要预防微风引起的潮湿，也需要预防飓风引起的漏雨。利用木质墙板来尽量减少预制板墙的使用。不使用地毯，只使用分离的垫子，大部分垫子都较小。

为春天、夏天和秋天的使用而优化。为冬天偶尔使用供热。使用电能踢脚板供热，而不是集中供热。它的单日使用成本较高，但资金投入低很多。

让噪声局部化或最小化。

- 为女孩卧室、男孩卧室和主卧室提供独自朝外的门，这样早起的人就可以溜到海滩上去而不影响睡觉的人。
- 将小屋划分成卧室区和公共区，将卧室、浴室以及通向它们的大厅从公共区域里区分出来。

隔离男孩的噪声。将男孩卧室放在卧室区的最远端。

设计随意的、不张扬的、休闲的内部装修。所有墙面使用护墙板，而不是涂料或墙纸。在起居室和塔楼等光线最刺眼的地方使用深色护墙板和踢脚板，厨房、餐厅和公共房间、男孩卧室也是这样。在所有地方提供暗的、安静的、清凉的感觉。在其他卧室使用浅色护墙板，制造欢乐的气氛。在大厅采用北极白的镶板，这样就不需要获取日光。

14个床位。4人睡在男孩卧室，4人在女孩卧室，2人在主卧，2人在客卧，2人在起居室，

2人在塔楼。在起居室提供2张沙发，适合睡觉。在塔楼提供2张可变形或可储藏的床。女孩卧室和男孩卧室各提供2个折叠铺位和2张永久的单人床。

22把餐椅。在小屋中放3张餐桌——两张8人桌，一张6人桌。

不装吊顶。出于经济和视觉效果的考虑，除了浴室和起居室外，都不装吊顶。房梁是doubled 2×12s on 4' center，设计能承受1英尺的降雪。屋顶由舌榫嵌入凹槽的2×6的梁构成，上面铺有绝缘垫衬，然后再铺上涂了沥青的防水层，上面再使用白石材反射热量。内部可见的部分只是上了清漆。

推论：不装吊顶使隐藏电线问题变得复杂。

优化塔楼楼梯占用的面积。为了将塔楼朝前推，楼梯必须位于小屋的前面，而前面的空间是很宝贵的。起居室将是前面唯一的公共房间，所以楼梯装在那里。

如果起居室希望让朝南的视角最大化，而且要通向其他朝北的公共空间，楼梯就必须靠东面或西面的墙。东墙有景观窗口、灯光和微风，所以楼梯将靠西墙。

似乎节省空间更好的办法是让楼梯占用正方形的空间，而不是长方形的空间，因为长方形的空间使整个起居室变得狭窄。所以我们使用了旋转楼梯。盐末撒在钢材上会很明显，所以楼梯选择木质的。既然楼梯是必需的，就将它装饰得像雕塑。

为采光控制而设计屋檐。屋檐必须有4英尺，以便从当年9月到来年3月之间允许中午的日光进入前面的房间，但在3月～9月之间则不会。

考虑正面

机会和约束条件让小屋的最大宽度可以是55英尺或660英寸。既然要充分利用海风和海景，下面就是关键的设计规划。

起居室。起居室对海景和海风的要求最高。很明显它将占有相当大的一部分空间。粗略地决定下，它会有16英尺宽。除前窗和门外，起居室还可以有侧窗，所以我们将起居室放在小屋的东南角，以利用东南面的景观，即恐怖角河口和船。

西露台。在小屋的西端设计一个狭窄的露台，可以直接走到海滩，吹到主卧的海风主要通过它，同时它也提供一条直接通道，让人从海滩走到内部的淋浴间，这样浑身湿漉的人不必穿过小屋。

卧室与餐厅-厨房。由于采用不装空调的决定，因此卧室的微风就变得非常重要。卧室直接能到海滩是次要的，但考虑到我们的孩子的个性，这可不只是个细节。所以卧室移到了前面，等分了小屋正面其余的部分。

女孩卧室与男孩卧室。这些是卧室中优先级最高的，因为它们在每次海滩旅行中都会使用（客人房间就不是这样），也可以作为孩子们的书房。

客人房间。主卧主要解决睡觉的需求。客人房间也可以作为书房，所以它赢得了正面的位置。

布置男孩卧室。为了隔离噪声，男孩卧室安排在小屋的西南角。一张床放在正面窗户下面，另一个铺位靠西面的墙，与前一张床重叠。如果另一个铺位靠北面的墙，可以很高（有探险的意思），接近房梁。衣橱就靠东墙放。房间最小的宽度等于门的宽度加上床的长度。

布置女孩卧室。如果不是完全放在上层铺位下面，较低的床就会感觉不那么局促。将一张床放在窗下，一个铺位靠东墙。另一个铺位也放在东墙，共用一个梯子。另一张床靠北墙，衣橱靠西墙。房间最小的宽度等于门的宽度加上床的长度。

布置客人房间。这间房间只需要一张双人床。它不需要通向海滩的门。最小宽度等于床的宽度加上床边的通道。房间的深度足够，所以将衣橱放在靠北墙的位置。

小屋的尺寸

平方英尺。可用经费决定了最多可以建2 000平方英尺，因为有许多窗。

房顶结构和房间深度。在两端起支持作用的均匀承重的横梁，其偏差与长度有密切的关系：

$$d = k\, l^4 / w^2 t$$

其中w是木材的宽度，t是厚度，l是有效长度。有效长度会因支撑点之外的支架而缩短。基于夏天和冬天日照的角度，我选择了4英尺的屋檐。全部计算表明，对于双重2 × 12s on 4' center木料，最大水平长度是16英尺。这确定了前面卧室、起居室和塔楼的深度。

餐厅-厨房的尺寸。如果主卧有一个门直接通向厨房，就像有一个门直接通往大厅一样，显然会有好处。于是厨房西墙尺寸就变得很关键，它的长度必须至少是一个工作面的宽度加上炉子的宽度，再加上冰箱的宽度，再加上卧室门的宽度，再加上炉子和冰箱之间的工作面。

我选择让厨房的宽度大致等于小屋背面的宽度，这样比较简单。这使得厨房的工作区很大。这也需要为屋顶横梁添加一些组合板条，以支持17英尺的跨度。

设想的开始

碎浪屋顶。开始我计划让屋顶轮廓线呈现碎浪的样子，如图21-1所示。

Cogswell强烈反对："我的建筑学教授告诉我们，'如果你们可以设计一个屋子抵挡风雨，孩子们，你们就做得很好了。'也许你可以在不伦瑞克县（Brunswick County）找到一个承包商让那个水槽不漏，但我怀疑做不到。"我听从了他回到基本样式的建议。

Brooks设想的无特色的塔楼屋顶。我为塔楼设计了一个完全无特色的屋顶。Cogswell进行了重大改进，如图21-2所示。

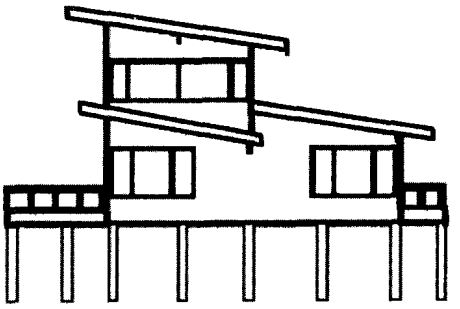


图21-1 Brooks建议的屋顶

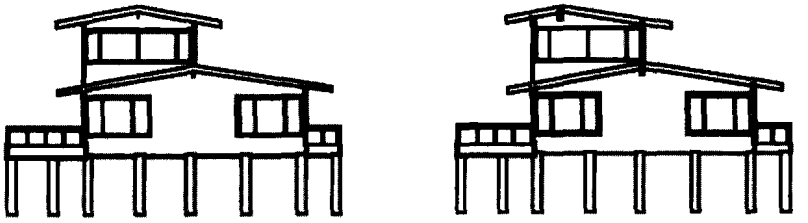


图21-2 Brooks和Coswell设计的东面高处

在设计之后，构建之前的设计改动

在底楼设计两个室外的沐浴间。为了不让沙、盐和湿漉的泳装进入小屋，我们设计了两个底层的外部淋浴间，换泳装将在那里进行。我们为每个淋浴提供了充足的换衣空间，可以容纳多人，这样父母就能帮助小孩。

用衣橱替代原来计划的大厅里的沐浴间。这部分空间变成了一个大的家庭日用织品橱和全高的杂物橱，有12英尺宽，它带有一个架子，下面存放一张可折叠的床，上面放纱窗。

将吃饭的区域从餐厅移到厨房。模拟研究表明这样让提供食物更方便。这使得餐厅变成了会客、工作、游戏和解决问题的区域。

将底层的储藏室扩大一倍，从8' × 16'扩大到16' × 16'。

将主卧的纱门换一个方向开。当浴室的窗打开时，纱门就打不开了——这是在建造时发现的设计错误。

在框架和外墙完成和初次入住之后的设计改动

决定不在卧室区域和公共房间之间建造隔断。

安装嵌入的斜线撑杆。起居室东墙和西墙上的撑杆，客人房间北墙上的撑杆，以及男孩卧室西墙上的撑杆提高了在大风中抵抗平行四边形歪斜的能力。这项工作在框架建好之后，安装墙板之前进行。

安装飓风防护固定夹。框架和外墙承包商没有安装规定的垂直的拉杆，所以我们换上了这

些固定夹，以便在大风中保住屋顶。

在露台安装水龙头用于冲洗。

为前面大的露台提供一个雨篷。雨篷为全部或部分露台遮阴。我们使用了定制的雨篷，针对高速路上70英里/小时的风速而设计。很久以后，我们用一个固定屋顶代替了雨篷，它向外伸出一半，让坐着的人可以选择晒太阳或待在阴凉处。

替换并扩大塔楼的窗户。飓风Diana（1984）吹走了所有原来的塔楼窗户、玻璃和窗框。我们将前面的多块小窗玻璃换成了两大块窗玻璃。这使得景观尺寸和效果都变得更好，防风能力也更强。

在西墙上加一个门。这让北面的小盥洗室既可以与主卧在一起形成一个套间，又可以成为公共卫生设施的一部分。

做了一些可移除的胶合板百叶窗。我们用这些百叶窗挡住小屋迎风的两面，以抵御冬天的风和飓风。

淘汰了多余的声音隔离（减少了墙的厚度），这面墙在起居室和客人房间之间。这项改动在框架建成之后，墙板完成之前。Brooks奶奶在1973年去世了，所以这间房间就变成成了客人房间，而不是奶奶房间，不再需要特殊的声音隔离。

将前面露台的东面栏杆换成长椅，专门为了享受海风和夕阳而设计。

安装了一个基槛（1997），以稳定固定桩。

在男孩卧室的西墙下面安装支撑（2000）。原来规划在西露台的边上打桩，而不是在西面的承重墙下，这是结构设计的错误。

评估（在37年后）

乐事

塔楼。塔楼后来成为了一间很好的书房，地平线的景色可以放松眼睛，能够看到沙滩上的活动，东南面可以看到大海水道上行船的景色，东北面可以看到河中行船的景色，还有很多灯光。《人月神话》的许多内容是在那里写成的。

开放规划。省掉原来规划的厨房和大厅之间的分隔，确实增加了乐事。这样一来，扩大了视觉空间，让人在进入小屋时更容易看到里面的人。同时也为主过道带来了日光和海风。厨师可以通过女孩的卧室和它打开的外门看到大海——这极大地提升了士气。彩色的卧室门成为了主要的装饰元素，大厅的白墙则很适合挂图。

旋转楼梯。橡木旋转楼梯是视觉上的亮点，像一座雕塑。在白天，它的轮廓映在前面的玻璃墙上。

设计合理。我们的建筑顾问Arthur Cogswell有一次笑称“View/360”是“该死逻辑的海滨小屋”，就因为这里介绍的详细理由和与他沟通的那些理由。我从不清楚他在拿什么做比较。当我在那里时，这种合理性就是我个人的一件乐事。

外观。从外面看，小屋很有趣，但不算美——功能决定形式（见图21-3）。Cogswell的不对称塔楼屋顶是正确决定，它似乎在向前跳。

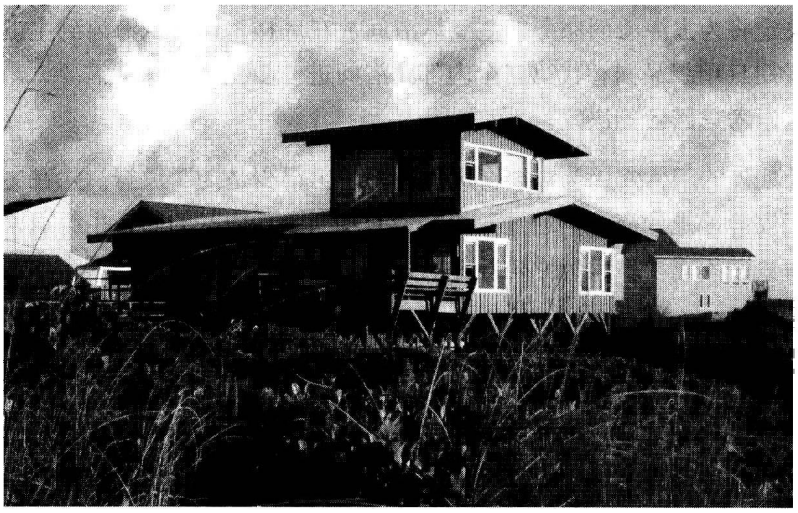


图21-3 从东南面看“View/360”小屋

实用性

目标实现。小屋很适合居住。

设计改动。在最初设计之后的那些改动已经证明是很大的改进。

省掉一面墙。省掉原来规划的大厅和厨房之间的墙，这使得大厅为厨房提供了最大的工作面。菜品以自助餐的形式提供，这是在设计时没想到的方便之处。多名厨师可以更方便地工作。

塔楼上的卫生间。在塔楼上放一个卫生间使它成为了一个独立的卧室套房，这是未曾预料到的好处。

餐厅。在原来的餐厅提供了一个独立的会客、交谈、工作、游戏区域，更适合居住。实际上原来设计作为厨房附件的部分，变成了第二起居室。它有时候也作为第二书房，虽然它不是隔离的。

主卧。主卧成为了另一个未曾预料到的第三书房，能看到湿地的景观。

容纳群体。虽然没有明确地为群体设计，但小屋可以容纳最多25人的周末人群，可是不能超过，这是基于我们成功和失败的经验。容纳25人时，睡觉（包括折叠床和地铺）、就餐、浴室和和会客空间比较紧张，但还能对付。

家庭规模。小屋现在（刚好）容纳全部家庭成员的聚会：我们的3个孩子、2个女婿与儿媳，9个孙辈小孩和我们——和当时设计的家庭比有了很大的变化。

厨房。厨房很容易容纳一组厨师，但一个厨师会觉得水斗离工作面-炉子-冰箱有点远。

露台。小屋西端的露台是一个大错误。它占用了42英寸的关键的精打细算的资源，即可建屋的海景空地。这个露台很少使用。这块空间用于其他用途会好得多。

后来我们决定将为去海滩而服务的主要沐浴间放在底楼，这使从海滩到主卫的外部过道变得没有必要了。用日用织品橱替代第二个内部沐浴间进一步减少了这种需要。当决定建造沐浴间时，我应该重新考虑露台的决定。

主卧。主卧朝外的门常常因为需要海风而打开，但很少是为了通行。住在主卧里的人可以通过厨房和后门安静地走到海滩。通过西面的窗户和两个朝内开的门，主卧通常就能得到足够的海风。缺乏海景并不重要，因为它是睡觉的地方，不是起居室。如果露台取消掉，这扇门也可以取消。

主浴室。主浴室朝外的门大多数时候是开着的，为整个屋子通风。如果没有露台，可能会想要一个上下部分可分别开关的门，下半部分固定关上。

隐私。考虑到对话的私密性，以及难以找到一个避开整个屋子噪声的地方，起居室、餐厅和厨房的开放性是不利的。塔楼在视觉上和社交行为上是隔离的，但不能隔音。

牢固性

小屋已经抵御了3次飓风的直接袭击和其他各种暴风雨。

- 1978年的暴风雨。我们失去了塔楼的组合屋顶（柏油浸透的织物单体壳）。风和伯努利效应将它掀起来，抛在了后院。
- 1984年的飓风Diana。风眼最近时不超过10英里，当地最大风速达到135英里/小时，是南风。大风掀起了除塔楼屋顶之外的所有组合屋顶，整个抛在了后院。屋内降雨达到16英寸。风暴吹坏了塔楼所有的窗户玻璃和窗框，并将床垫、地毯和灯吹到了湿地。
- 1996年的飓风Bertha和Fran。风眼最近时不超过10英里。除了一扇百叶窗和一块窗玻璃，没有造成其他破坏。

桩上差异巨大的承重导致了在沙土上不同的沉降。设计没有考虑到不同的沉降。这是奇怪的疏忽，因为在沙土桩上的每幢小屋都要遇到这个问题。塔楼南面两个角下的桩承受了额外的重量，因此沉降得厉害一些。非承重墙下面的桩沉降得少一些。这个问题在起居室大的双扇门中央尤其糟糕。门的导轨中间上拱，导致移门不能正常关上。我们在1997年安装了一块4" × 6" × 16'的地下基石，拴住塔楼南面墙下的三根桩，这样将来的沉降会少一些，并且会均匀沉降。

最西面一排桩打在露台下面，而不是在支撑栋梁和屋顶重量的西面墙下。地板梁在无支撑的重量下发生了弯曲。28年后，不得不在那面墙下加一些桩。很清楚，这是一般的疏忽。Cogswell和我都没有仔细对照主地板平面图的桩平面图。

平开的窗户是一个错误。收集微风的功能像计划一样。虽然窗户是木头的，但铰链是钢的，小屋朝海浪的两面每5年要更换一次铰链，背风的两面每15年更换一次。在第35年，当老的窗框坏了的时候，我们将大部分的平开窗换成了双悬挂窗。

什么因素导致了平开窗的错误？没有充分考虑长期项目的维护重要性，也没有充分考虑所有的建材。

如果我“废弃一个计划”呢

假定我现在针对这个位置和1972年的家庭情况设计这间小屋，基于现在的知识，我的设计会有什么不同呢？前面的“评估”小节详细介绍了各种较小的漏算和错误，下面是大的教训：

来自于前面文章（见第10章）的第一大教训，就是更加注意精打细算的资源，在这个例子中，就是朝着大海一面的每一英寸。现在既然明白了这很重要，我会研究边线留出距离的要求细节（弄清楚屋檐算不算），我的设计会利用每一英寸，甚至会打破平方英尺的预算。

第二大教训（见第11章），就是要注意到早期在规划中添加了底层淋浴间，这消除了从外面进入主浴室的迫切需要或约束条件。因此我会移除西面的露台，并重新分配它占用的42英寸朝海的一面。

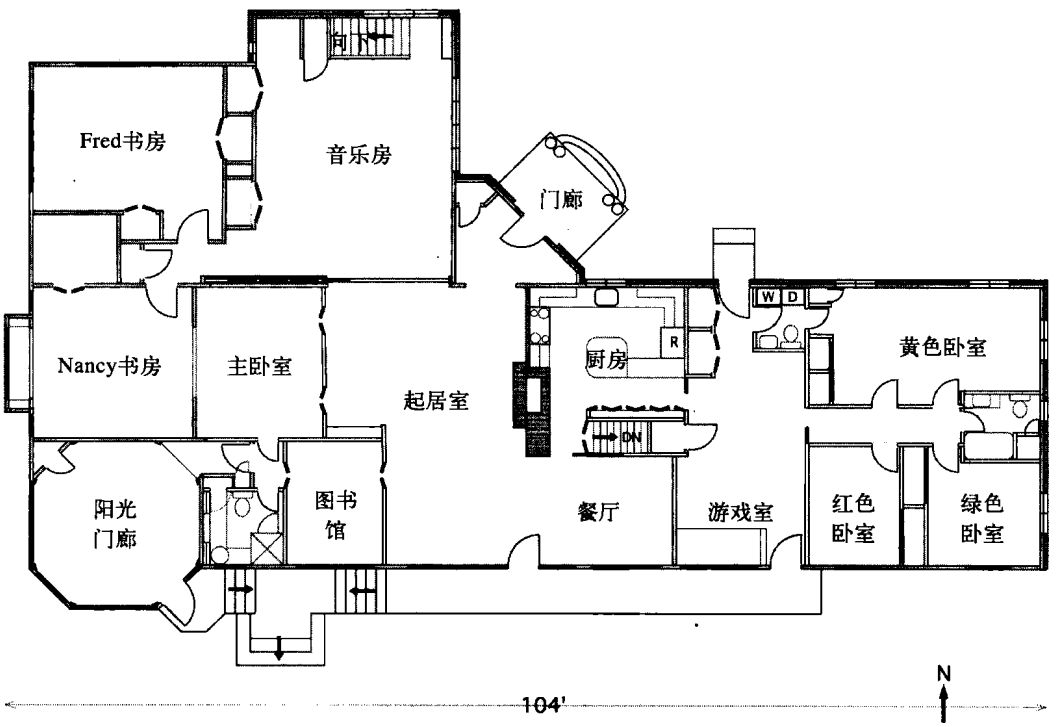
学到的一般经验

这里学到的局部领域的经验可以广泛地应用于所有实际的设计项目，不论是硬件、软件，还是建筑：

1）非常仔细地检查你的专业建筑师或架构师的工作，并询问理由。即使是诚实的、有能力的、尽职的建筑师或架构师也会犯错误。

2）在建造过程中从头到尾经常检查。即使是诚实的、有能力的、尽职的建造者也会犯错误。

3）仔细考虑所有的维护方面。任何成功的设计都需要你维护很长时间。



1991～1992年增加厢房后屋子的平面图

第22章

案例研究：增加厢房

实际上，在语义丰富的任务领域，架构总是可以看成是设计过程的原型。

——Herbert Simon (1981)，《The Sciences of the Artificial》

亮点和特性

为什么是这个例子？对于这个设计，我们有大约235页的设计日志，记录了当时的设计问题，包括优点和缺点，以及在60多个月中所做的决定。这个例子展示了第3章中讨论的设计与需求发现之间的相互影响。

大胆的决定。推迟预算约束条件，针对功能设计，然后是价值工程。这些决定是在设计工作进行到一半，还没有什么能工作时进行的。

大胆的决定。将主卧室移到公共空间和半公共空间的中间。这个决定是在设计过程较晚的阶段做出的，当时一个发生频率较低的用例揭示了之前未注意到的需求。当时这个决定似乎意味着基本上放弃屋子的东面。随着后来家庭的发展，拥有这样的“旅馆”部分变得非常有用。

关键的决定。向邻居购买5英尺的条形地块，以解决难处理的设计问题。这个故事在第3章中介绍过。

阶段划分。整个屋子的重新建模分两个阶段进行，以简化我们的设计和监理任务，本章和第23章将详细介绍。

充足的设计时间。设计直到我们满意为止，不受任何建造进度计划目标的限制。在这件事里，设计超过了60个月（中间有一些明显的中断），建造只花了9个月。

背景介绍

位置：

北卡罗来纳州教堂山市（Chapel Hill）Granville路413号

屋子朝向正北，所以描述将用东、南、西、北的方式。

所有者：

Frederick和Nancy Brooks

设计师：

Frederick和Nancy Brooks

Wesley McClure（美国建筑师协会会员）和Alex Jones（美国内部装饰协会会员）有时提供建议。

建筑图纸由另一位绘图员完成。

建造者：

另外加上（承包商Stanley Stutts和项目主木匠Gary Mason）

日期：

设计从1987年至1992年

建造从1991年至1992年

辅助网站：

在设计过程中，Fred和Nancy保存了一份详细的日志，大约235页，记录了设计问题、困难、想法、朋友和专家的意见、决定等。Sharif Razzaque将日志中比较重要部分编成了一张概要图，展示了决策树，但没有包含决定的详细理由。这棵树可以参见本书的网站：www.cs.unc.edu/brooks/DesignofDesign。

背景

1964年～1965年

原来的屋子（见图22-1）是1960年造的，1964年购入，选择它的主要原因是这块地有大片树林，并与小河相邻，位置也很方便，尽管屋子有明显缺点。最严重的一个设计缺陷就是通行方式，特别是餐厅与厨房之间的瓶颈，所有从东到西的通行都要通过它。而且，屋子的每个房间相对于它的功能来说都感觉小了。

我们在1965年搬入的时候，两个儿子一个7岁，另一个3岁，女儿才6个月大。为了靠近孩子们，我们将红色卧室作为主卧，另外3间东面的卧室留给小孩。原来的主卧在西面，在1965年～1986年间作为客人套房。

1972年

我们将地下室改为一间房间，将大儿子移到那里，二儿子不久也移到那里。这让我们能够移除两间东面卧室之间的分隔和壁橱，得到一间更大的卧室，即黄色卧室，给女儿用。绿色卧室变成了Nancy的书房。

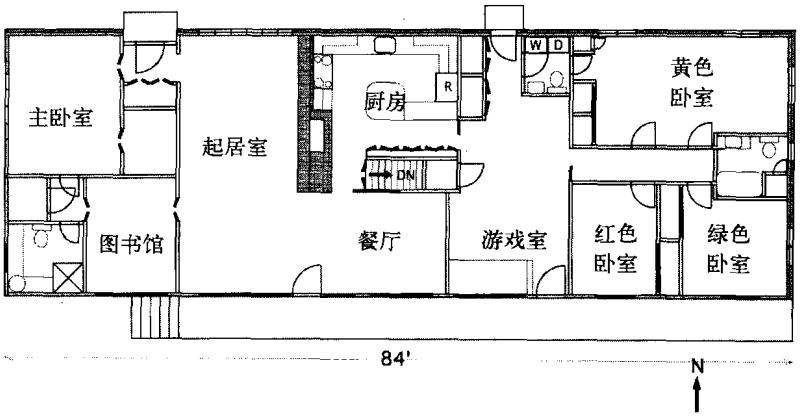


图22-1 1987年的主层平面图

1987年

我们的女儿在1986年从大学毕业了，成为一名军官。我们的儿子当时在读研究生和实习，一个儿子结婚了，没有小孩。孩子们长大了，我们有了更多时间做其他事情，因此感觉需要更多的特别空间。Nancy Brooks从20世纪60年代中期一直在家中教授小提琴，现在她可以教更多学生了。

Nancy鳏居的父亲Joseph Greenwood博士刚搬过来，需要和我们长期住下去，他住在西面的主卧（客人）套房。Nancy继承了她父母的大钢琴。这补充了我们拥有的一架钢琴，可以演奏双钢琴的音乐了。

从很多方面来看，这幢屋子都需要一次30年大修。Fred的书房在地下室房间里（从地面可以进入），那里曾是男孩宿舍。考虑到我们的年纪，似乎最好能够支持只用主层的普通户内生活，有一天会需要这样的。

在前面的几年里，我们曾试着看看500平方英尺的增加面积如何能够让屋子更宽敞，最好是交换功能，让每个房间的功能有更大的空间。这些设计努力并不成功。

所以现在我们开始非常认真地设计增加面积。

目标

最初目标

- 改进通行方式。
- 让每间房间有更大的空间。
- 建造一间足够大的音乐房，能容纳2架大钢琴、一架小型管风琴、一个弦乐八重奏，并在八重奏的周围留出1英尺的走道，便于指导。这间房还必须存放音乐文件。这样的尺寸可以让两架大钢琴移出起居室，它们在起居室里占据了北面，从前门进入只有一个狭窄的入口。音乐房应该能够容纳举办小型学生音乐会，包括作为听众的父母。它最好有

一个独立的入口。

- 将Nancy的书房从东南角的绿色卧室移出来，从它能方便地走到音乐房和Fred的书房。我们需要扩大它的面积。
- 从Fred的书房上楼能到主层。
- 提供更多的功能空间：房间和空地。
- 添加一个足够大的前门廊，可以放一个门廊吊椅和其他椅子。
- 提供一个装玻璃的后门廊（这就是阳光门廊）。
- 扩大厨房并使之现代化。
- 扩大餐厅。
- 当有人从车道开车过来时，能更明显地看到主入口（见图22-2）。
- 改进设计的美观，特别是外观，也许可以换一个更有趣的屋顶。
- 增强庭院和场地，至少不要损害。
- 保留西南面、东南面的树和北面的花。
- 在屋内能看到庭院和花园的景观，特别是从公共房间。

后来发现的目标

- 更好地容纳我们指导的学生进行开会。聚会两周一次，大约40个人。
- 提供地方放大约40件外套，针对参加音乐会的人和学生开会。
- 为现在放在租用储藏室的家庭用品提供储藏空间。

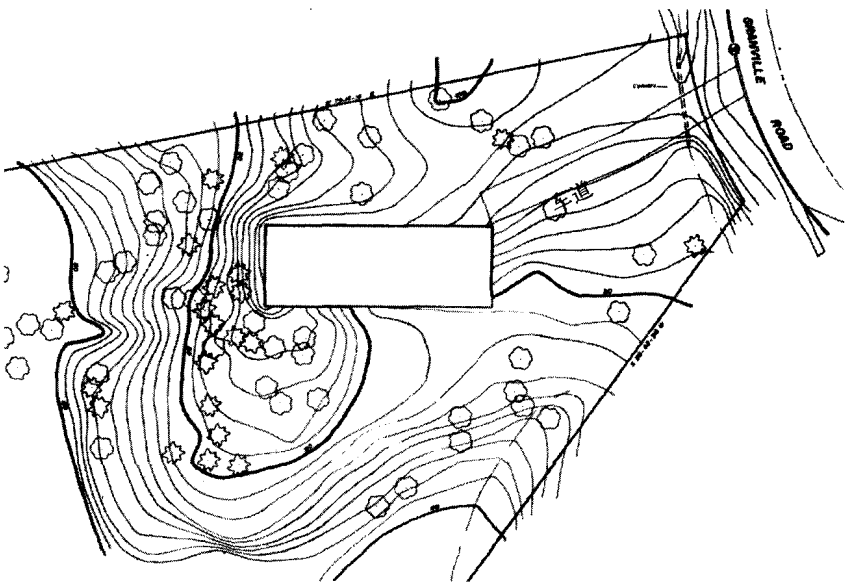


图22-2　屋子地块的北部

约束条件

原有的结构。平面图、布局和原有结构的朝向决定了重新建模的范围。

位置。北面的地产边界线和15英尺的留出距离，17英尺的阴影投影（shadow-casting）留出距离要求，这些都是扩展的约束条件。

树。后院的一棵大黑橡树是这个地块的主要特征，我们希望保留。

地块。西南的地势相当陡峭地下降，从原来屋子的西端开始。

预算。我们的目标是10万美元，预期增加的成本是每平方尺100美元。

非约束条件

预算。我们的10万美元的目标不是绝对约束条件，因为购买时的抵押贷款已经全部还清。

转卖价值。不需要考虑对增加部分的投资是否会增加转卖的价值。根据预期寿命和我们不打算搬迁的计划，预计我们可以花30年时间分期清偿增加部分的费用，再卖出这幢屋子，那时候又需要再次大修了。

土地面积。土地很充足，这块地的面积超过了1.5英亩。

时间和工作量。我们有充足的设计时间，并愿意在设计上投入很多工作量。

事件

在设计过程中，一些事件改变了设计：

- Greenwood博士在1988年下半年去世了。
- 建筑师请的绘图员负责绘制建筑图纸，把基础图纸和主层图纸的定位弄错了。这一差异在基础浇注之后才发现。修复方案就是将主层向西和向北增大1英尺。本章首页的插图展现的是建造时的情况，而非设计时的情况。

设计决定和迭代

考察

将屋子重新翻修一遍。进行大规模内部改造，让原有的卧室成为公共房间（音乐房、起居室，也许还有Nancy的书房）的厢房，将主卧和客卧放到老音乐室的位置。**好处**：这会使主入口和学生入口的通行变得容易，而且我们也许可以添加一个书房，作为东南面的扩展。**不足**：这是成本很高的改动；我们需要在西面有更多的浴室空间；我们会浪费掉壁炉；从餐厅到起居室的通行与其他穿过厨房的通行将分离开来。我们很快放弃了这个选择。

东南厢房。在屋子的东南面建造新的主卧套房，包括新的绿色卧室。为了保留屋子东南面的白橡树和红橡树，也为了从车道能够进入房间和后院，我们放弃了这个想法。

McClure设计的公馆（见图22-3）。**好处**：音乐房（北面的部分）变成几乎完全独立，而且很容易从车道进入，与屋子的隔音也很好。起居室（南面的部分）有很好的景观，能看到花

园和院子。空出来的原来的起居室变成围绕壁炉阅读和谈话的地方。**不足**：南面的部分要牺牲掉那棵漂亮的、四根主干的黑橡树。在举行小型音乐会时，我们不能利用起居室的空间来扩展音乐房。

南面的部分很快放弃掉了，所以导致了重新设计起居室，用以包含吸引人的炉边空间。

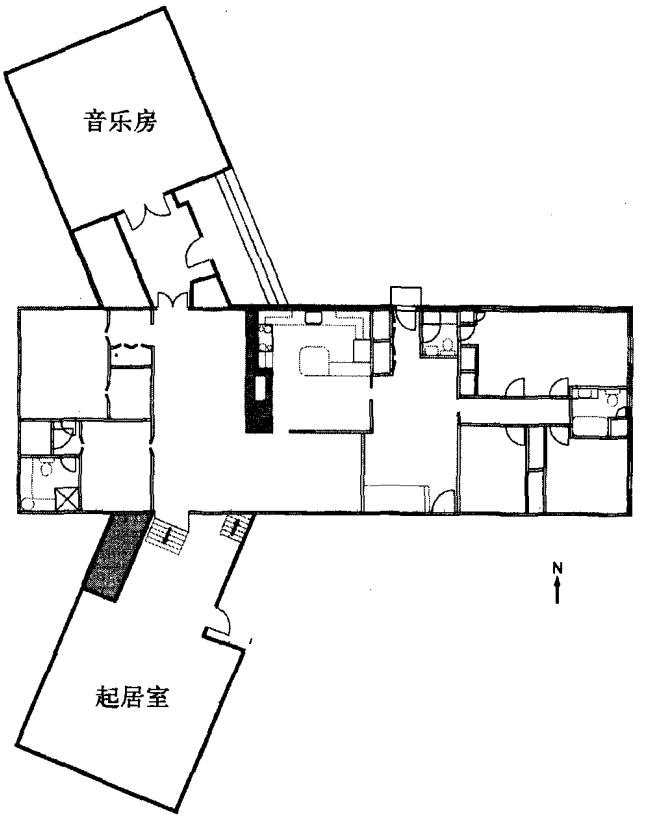


图22-3　McClure公馆的草图

北面的设计保留在设计主线中，经过了许多形状和配置的考察，最后成为北面的厢房，包括音乐房、Fred的书房、门厅和前门廊。

分割设计问题

在设计过程中，我们逐渐明白屋子的重新设计可以划分成3个几乎独立的问题：

- 旧房东部：卧室部分，还有可能包括游戏室。
- 中部：厨房、北大厅及其壁橱、游戏室、洗衣房与卫生间，地下室楼梯。
- 旧房西部：音乐房、餐厅和西面卧室套房，以及新加出来的部分，即新房西部。

这种划分被证明是自由的，所有后来的设计都基于这种划分。

阶段。很早的时候，我们决定有两个设计和建造阶段，中间隔几年，主要是为了让设计和监理任务可管理。阶段I是老西部-新西部的部分，阶段II是厨房和游戏室的部分。阶段II作为

单独的案例在第23章讨论。

东部

东部进行了各项重新设计，目的都是为了让主卧室有自己的沐浴卫生间，并为Greenwood博士提供一个舒适的套间。一些试验性的考察考虑了将地下室楼梯移到红色卧室，或放在它与游戏室之间。在Greenwood博士去世之后，除主卧外还需要一个卧室与浴室套间的需求消失了。因此在设计过程中，在我们将主卧移到西部之后，我们放弃了东部的重新设计工作，让它保持原样。

它作为客人套间，只在有访客时供暖或制冷。Roger的家庭现在有3个小孩，Barbara的家庭有4个小孩。不论哪个家庭来访，都会住在这个套间和地下室的房间。

西部一半的功能安排

最多的考察是考虑旧房西部和新房西部的功能安排，它们被视为一个可分配的空间，虽然原有房间的墙会尽可能保留。

这些考察发生在决定将主卧保留在西部之前，旧的主卧被视为可分配空间的一部分。所以所有早先的思考仅考虑了音乐房、起居室、餐厅、书房、Nancy的书房和Fred的书房。本章首页的平面图在这个阶段是会引起读者误解的。

既然排除了在南面扩展，我们考虑在北面、西面添加，或在两面同时添加。功能分配的考虑是：

- 音乐房在西面，起居室在北面。
- 音乐房在北面，起居室在西面。

方式变化：忘掉预算是设计约束条件

在进行这些考察时，显然预算上的约束条件转化到1 000平方英尺上妨碍了我们思考。所以我们决定先让设计满足目标，然后再进行成本工程，或者在觉得值得的时候再决定花更多的钱。这极大地解放了我们的思维。

我已经在计算机图形系统的设计中宣传这种方法好几年了。我发现要得到一个有成本效益的应用系统，最好的方法是先设计有一个有效益的，然后再削减它的成本，而不是先得到一个便宜的，然后再扩展它直到能用。我花了太多的时间才在屋子设计上想到同样的方法。

新发现的需求：在哪儿放置外套

1990年11月，我们进行了检查试验性的设计，针对它执行各种（临时准备的和没有记录文档的）用例。我们比以前更详细地执行用例，因为设计更确定了。我们执行了两周一次的聚餐场景，有30～40名学生，我们是他们的技能导师和房主。

当客人在冬天到达时，我们将他们的外套放在某处。放在哪呢？门厅的外套壁橱不能容纳。

放在乐器上？放在我们的书房？它们现在是放在哪里？放在客房的床上，与起居室相邻。哦，这个房间在这个设计里已经没有了。

一个解决方案是扩大外套壁橱，幅度相当大。另一个解决方案是在西部保持那间客房，我们搬进去作为主卧，再重新设计东部。毛成本：朝西扩展得更大，至少是那间客房现在的宽度。净成本：毛成本减去东部未发生的成本。

问题。如果朝西的扩展超出主卧室，那么扩展部分必须与主卧室集成，以满足睡眠空间道德准则方面的需求。在卧室和窗户之间没法加一扇可以关上的门。

推论1　如果我们把Nancy的书房放在朝西扩展的部分，那么这种集成就不是问题。其他房间都不能放在那里（Fred经常在Nancy睡觉时学习，但反过来不是这样，所以Fred的书房不能放在那里）。

推论2　如果音乐房不放在西面的扩展部分，它就应该放在北面的扩展部分。这让它与主要的生活区域隔离开来，而且能够方便地从车道进入。

功能安排的会合

事情之后就很快安排好了。

起居室。为了满足音乐会的要求，最好是有时候能够让起居室和音乐房成为一个整体，但并非所有时间都要这样。这个问题是通过一个12英尺的移门（四扇）解决的，它可以移入主卧北墙外面的一个新容器里。

起居室的扩大是通过合并原来的门厅、原来的门厅衣橱和原来的客房衣橱来实现的，原来的这些功能都放到新的厢房中。当然，将两架大钢琴从起居室移到音乐房使得起居室的有效面积大为改观。

Fred的书房。它现在的合理位置是北面厢房和西面厢房之间的角落。这样也为新主卧和其壁橱留出了空间，新主卧的壁橱不在Nancy的书房了。北面大厅还放置了复印机，在音乐房、Nancy的书房和Fred的书房中间的位置。

阳光门廊。我们想要的南面门廊很漂亮地放在了新西厢房的西南角。原来我们设想它是一个地面上的门廊。当我们发现主层到门廊的台阶将占用门廊较多的面积时，我们决定将门廊保持在主层。同样的，用例检查引导我们为它装上了玻璃，有许多可以打开的窗，而不是为它遮光。户外的门廊使用率会比较低——冬天太冷，夏天太热。

前门廊。前门廊历经了几次修改，最初它占据了北厢房的整个东面。在这个项目过程中，我们保留了McClure的北面部分的最初的斜线朝向，直接面向从车道到屋子的小路。它造得足够大，能容纳两个面对面的门廊吊椅。

门廊45°朝向所形成的人字形屋顶解决了两个问题。首先，它为屋子提供了一个明显的入口。其次，它恰好平滑了老屋子8英尺的屋顶和新厢房9英尺的屋顶，以及它们相应的屋檐（见

图22-4)。

图22-4 从东北面看重新建模的屋子的景色

地下储藏室。我们在较晚的时候发现了一个机会，就是可以在西面厢房下面设置廉价的储藏空间，因为地势下降得很快。通过适当挖掘和低成本修整，就在音乐房的下面得到了530平方尺的封闭储藏空间，一间小的工作室，以及放置新厢房机械设备的空间。这让我们能够放弃一直租用的远程存储空间，使用也更方便了。

地产边界线留出距离约束条件。第3章讲了一个最有趣的单项设计问题和决定的故事。音乐房的目标相当具体，使得它必须近似方形。这一点以及Fred书房的合理安排与法规要求产生了冲突，教堂山市（Chapel Hill）要求阴影投影留出距离（shadow-casting setback）是17英尺。花了大量时间进行设计修改也不能解决这个问题。我们最后向邻居买了一条5英尺宽的地。

餐厅。我们为餐厅设计了一个简单的朝南扩展。这使得餐厅的长边是南北向的，而不是东西向的。它需要一个人字形屋顶，同时也让屋子南边的露台感觉有些别扭。在估计了成本之后，我们放弃了这一设计，这纯粹是成本工程的一部分结果。

构建期间的改动

Fred的书房窗户。根据设计，Fred的书房西面的窗户有4英尺高，离地板3英尺。在制作窗框时明显发现，西面下降的地势意味着从这些窗口看到的景色只是树梢。所以窗户改成了6英尺高，离地板1英尺。这确实改进了这间房间的景色和感觉。这就是那种计划和立体图永远也不会揭示的问题，但虚拟环境模拟将在设计时就能发现这类问题。

管风琴壁龛窗户。儿子Ken Brooks建议在新管风琴的长凳背后装一个狭窄的窗户。这为音乐房提供了第三个方向的采光[1]。

评估——成功与未解决的缺点

现在离第一阶段项目完工已经过了大约17年，离第二阶段厨房–游戏室的重新设计已过了14年。我们没有一份“希望当初做得不一样”的清单。异乎寻常的长时间设计努力和对细节的仔细关注，在宜居性、功能和快乐方面带来了回报。当然，考虑到约束条件，并非所有的理想都能实现。

厨房到起居室的门。增加从厨房到起居室的门带来了最戏剧性的效果。这极大地改善了屋子的通行状况，并带来了通透的视觉效果。

主卧室。这张平面图确实有些奇怪，在主卧室周围有一圈其他功能。这样设计是因为不考虑转卖价值。很少有家庭需要那么大的音乐房、Fred的书房和Nancy的书房。

音乐房。这个空间很适合授课和个人练习，因为它可以是一个封闭空间。它很好地容纳了一架大的管风琴。它适于举办音乐会和年度研讨会，因为它可以与起居室合并。但它没有独立的进出通道。学生将鞋和乐器放在游戏室，然后走过厨房和起居室。

起居室。将两架钢琴移出起居室使它成为了一个新房间。通过壁橱合并而带来的空间扩大受到了欢迎。但是宽度增大能让人觉得屋顶更高一些就好了。照片和视频投影有点尴尬。没有合适的地方挂屏幕。

餐厅。这间房间仍然很拥挤。桌子可以扩展到起居室，补充的桌子也放在起居室，所以可以让许多人坐着就餐。

前门廊。两张面对面的吊椅创造了一个独立的谈话角落，我们经常使用它。主要入口现在很明显了，外观也有了很大改进。

新功能

后来我们发现，重新设计的屋子满足了一些我们从未想到过的需求。正如在前面的内容中提到的，这对于产品是常见的情况，并不是例外。

阳光门廊作为会议室。我们发现阳光门廊非常适合不超过11或12人的会议。

音乐房作为起居室的扩展，用于会议。最初的需求是起居室作为音乐房的扩展，举行音乐会。这个需求实现得很好。学生和他们的父母，一共大约25人，坐在几排椅子上，位于音乐房和向后扩展的起居室中。

没想到，后来我们发现相反的方向也是成立的。InterVarsity基督教联谊会（IVCF）的毕业会议通常有30～40人，但偶尔会吸引超过50人。这些会议的焦点是在起居室壁炉边的演讲人，音乐房可以容纳后排的座位。

露台、台阶和作为会议、就餐区域的庭院。南面外墙边的露台通过一些宽台阶可以下到地势较高的后院，露台上安装了一张可折叠的桌子。我们后来发现这是户外就餐和IVCF全体成员开会的好地方。台阶提供了很多座位。

学到的一般经验

1）在设计上花时间。我们在每平方英尺的设计上花了太多时间，如果设计师的时间是产品价格的一部分，那么这根本就不可能有成本效益。对于Linux来说，情况可能也是如此。对于OS/360，如果开始实现之前花更多的时间进行设计，我们可能会受益良多。我不认为产品的总成本会更高。

2）与主要用户进行多次长时间的交谈，向他们展示他们能够理解的原型。

3）执行大量的使用场景。

4）再次检查专业人士的工作，如建筑师、工匠和装修人员。确保你能理解这些工作，并确保这些工作是准确的。

注释

1. Alexander (1977)，《A Pattern Language》，主张确保每个房间的日间采光至少有两个方向，最好更多。我们严格地遵守了这一点，为Nancy的书房和Fred的书房加上了天窗（但有意没在主卧室加天窗），并在Nancy卧室与阳光门廊之间的墙上加了一个窗户。这样，每个新房间都有3个方向的日间采光。

重新建模厨房的虚拟环境模型

第23章

案例研究：厨房重新建模

如果你不能忍受热度，就不要待在厨房里。

——杜鲁门，美国第33任总统

亮点和特性

为什么是这个例子？这个简单的例子展示了设计工具的威力。图纸、计算机辅助设计（CAD）软件、按比例绘制的模型、全尺寸的仿真模型，以及虚拟环境（VE）下的查看都为设计带来了好处。VE和仿真模型带来了相互不可替代的价值。

大胆的决定。移动外墙。这个改动改变了设计。

大胆的决定。在厨房与起居室之间增加一扇门。设计这个门完全改变了屋子的通行方式。

天窗。两个天窗让黑暗的、朝北的厨房变成了明亮的、舒适的空间。

背景介绍

位置：

北卡罗来纳州，教堂山市（Chapel Hill），Granville路413号

拥有人：

Frederick和Nancy Brooks

设计师：

Frederick和Nancy Brooks
Wesley McClure（FAIA）和Alex Jones（ASID）提供了建议。

日期：

1995年～1996年

背景

这部分设计是对20世纪60年代的房子在90年代重新建模的第二个阶段。第一个阶段包括西面的厢房、一间门厅和一个门廊。这在第22章中进行了介绍。第二阶段安排在第一阶段过后几年，以便为第二阶段留出充足的设计时间，并对第一阶段的构建进行充分地检查。

目标

首要目标。我们的首要目标是对又小又暗、朝北的厨房兼早餐厅进行改造，将它扩大、重新安排，并增加亮度。

其他目标。按重要性递减的顺序：

- 改进屋子的通行方式，原来两部分之间的所有通行都要通过地下室楼梯旁边的狭窄过道。我们也需要让练习小提琴的学生从后门进来，从琴盒中取出小提琴，将琴盒放在游戏室，走到音乐房，再走回来。
- 将厨房的桌子移近朝向花园的窗户，以便有更好的景观。
- 安排空间，让厨师能够与坐着不干活的来访者交谈。
- 使厨房能够方便地
  - ■ 让一名厨师准备早餐；
  - ■ 让一名（矮）厨师进行一般的烹饪、烘焙和制作罐头；
  - ■ 让多名（最多3名）厨师准备大餐。
- 通过自助餐的方式方便地服务30～40名学生。
- 增加相当多的操作台空间。
- 安装一个更大的水斗。
- 设计一个步入式的餐具室。
- 保持外观舒适。
- 增加到屋子的“后门”入口的光亮，这也是家庭成员、学生和非正式访客的主要入口。
- 让后门与外部正面相配。
- 为沉闷的、砖砌的烟囱添加一个装饰性的壁画。
- 将杂物藏到橱柜里。
- 展示少量的玻璃制品。

机会

较小的家庭。由于孩子们长大了，家庭变小了。所以一般在厨房就餐会是两个人，偶尔是三个人，四个人已经是例外，而不像以前总是五个人。

游戏室中可用的空间。由于1992年在新的西厢房建造了一间音乐教室，以前由管风琴占据的5英尺×5英尺空间现在可以利用了，还有以前的高保真音响设备占用的2英尺×6英尺的间。

三足屋檐（Three-Foot Eaves）。受到Frank Lloyd Wright在20世纪60年代农牧场风格房屋的启发，这幢屋子的屋檐很深。

设计时间和工作量预算。这些方面基本上是没有限制的。

约束条件

用户高度。厨房主要用户的高度是5英尺1英寸。

建造预算。预算不紧张，但不允许进行较大的结构变更。

房屋外形。房屋的外形通过1991年的扩建（见第22章），已经变得很有吸引力，我们不希望破坏它。

原有的厨房。原有厨房的面积和形状（见图23-1）将决定新厨房的形状。

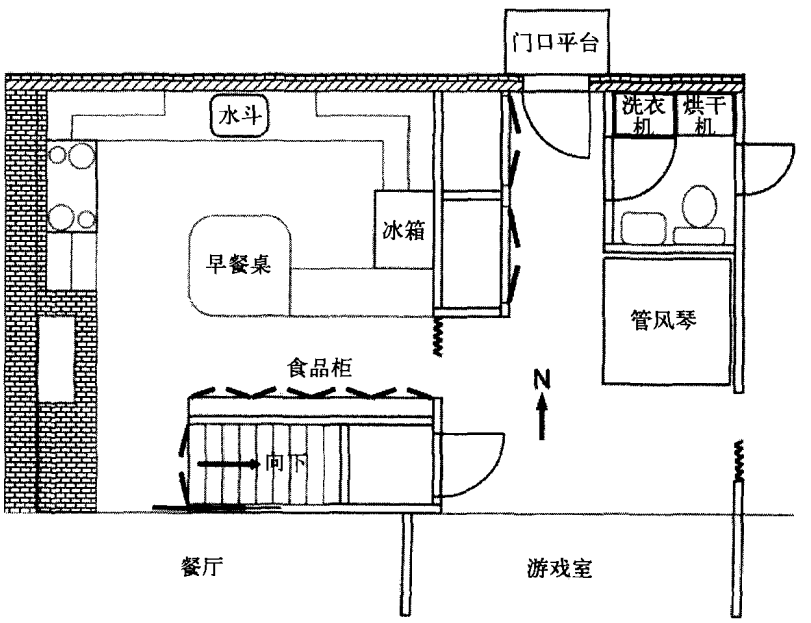


图23-1 重新建模之前的厨房平面图

砖砌的烟囱墙。这面墙有8英寸厚，限制了通行的方式。

地下室台阶。它们的位置不能改变。

游戏室朝外的门。这扇门的功能很重要，但位置可以改变。

后门。它开在砖墙上，所以移动它的成本很高。

原有的洗衣房/卫生间。这间房间不需要改变。

食品柜。我们需要保持食品柜的功能和容积，但不一定保持它的位置。

壁橱。北大厅一些壁橱的功能和面积必须保持，但不一定保持它们的位置。这是一个很有

趣的例子，说明了仔细研究原有使用场景的重要性。这项需求不能从先前的厨房设计经验中推想出来。这项需求的强大之处在于，它来自我们30年所遵循的特定使用场景，如果这些壁橱里的东西分散在屋子各处，就会带来极高的“混乱成本”。

结构上的考虑。地下室楼梯墙上的垂直结构构建支持着屋顶。

屋子其他在使用的部分。在建造过程中，其他房间将一直使用。

关键宽度预算的推理

从北到南需要的宽度

在早期寻找可行设计的所有尝试中，我们发现宽度是一个障碍。考虑以下目标：

- 窗户边的就餐区域；
- 从水斗处可以看到窗外；
- 从水斗处可以与就餐区域的人交谈；
- 容易穿过厨房东西向通行；
- 足够的操作台/橱柜空间；
- 炉子-水斗-冰箱形成短三角形。

试验性的设计

这些要求意味着需要设计一个岛状的水斗，朝向窗户和就餐区域。让带炉子的操作台背靠南面的墙似乎是必要的。

这样一来，从北到南的宽度必须考虑下面的部分：

- 餐桌（不少于30英寸）
- 通行通道（不少于24英寸）
- 水斗岛（不少于24英寸）
- 水斗和炉子之间的通行/工作空间（最初估计不少于36英寸）
- 炉子和操作台（不少于27英寸）

这些加起来是12英尺3英寸。而原来的宽度是12英尺。

餐桌坐4个人的要求可以通过让来访者坐在过道上来满足，这实际上堵塞了过道。但这是可以接受的，因为这只是偶然的情况。

但是仿真研究表明，从水斗到炉子之间的空间实际上需要44英寸，而不是36英寸，因为柜橱需要开门和抽出，炉子需要开门，洗碗机也需要抽出。所以总的宽度需要不少于12英尺11英寸。

另一些宽度解决方案

消除通行通道。消除通道，让东西向的通行经过水斗和炉子，完全干扰烹饪。这个设计被否决了，这意味着宽度仍需要12英尺11英寸。这表明我们可以：

- 收回食品柜和地下室楼梯的空间，将它们移到别处；
- 为就餐区域设计一个凸窗；
- 在屋檐允许的情况下，将整个厨房的北面墙外推。

在后面两种情况下，可以将食品柜移到别处，从而换来9英寸的宽度。

移动地下室楼梯？对移动楼梯的大量研究表明，唯一可行的替代方案是在屋子外面加一个旋转楼梯塔，连接到游戏室南面朝外的门。其他解决方案要么不能与原有的或合理的楼上布局相匹配，要么不能与地下室的合理布局相匹配。

楼梯塔的替代方案研究了几个月。最后它被否决了，因为造价高，而且不美观，虽然它能够解决通行问题。

对于设计过程来说，得出所有“移动楼梯”的替代方案都不可行或不可接受是一个关键的时刻，这极大地缩小了可能设计的范围。这个子设计是Simon“搜索树”过程的一个好例子，因为我探索了多种“旋转楼梯”的设计方式，在没有一种可行的情况下，回溯到设计树的上面一层，决定根本不移动楼梯。

是采用凸窗还是将整个北墙外推？外形仿真研究表明，将整个厨房北墙外推比加一个凸窗要好看很多，同时成本上大致差不多。所以我们选择将整个北墙外推。同时仿真研究还表明，外推18英寸或24英寸是美观的，因为屋檐有36英寸。

最终的宽度设计

有了外推得到的24英寸和移动食品柜得到的9英寸，宽度限制大大缓解了。水斗从24英寸加宽到36英寸，提供服务、准备以及存储空间。北面的通道加宽到39英寸。南面的通道加宽到44英寸。

长度预算的推理

长度的困难。南面墙的长度成为了一个难题。它必须容纳48英寸的炉子，在炉子左边和右边分别是一个工作台。西边的工作区域决定用作早餐和烹饪区域，所以它必须容纳一个微波炉和一个烤箱，总长度不少于36英寸。

东边的工作台是主要的一般烹饪和烘焙区域，可以拿到干的配料、调味品、搅拌器和其他烹饪所需的东西。仿真研究表明，操作台的长度最好有48英寸。

设计。南面的墙向东增长了18英寸，使厨房和游戏室成为隔离更好的两间房。将新餐具室放到位后的仿真研究表明，5英尺的（对角线方向）开门就足够了，而且有很好的视觉效果。

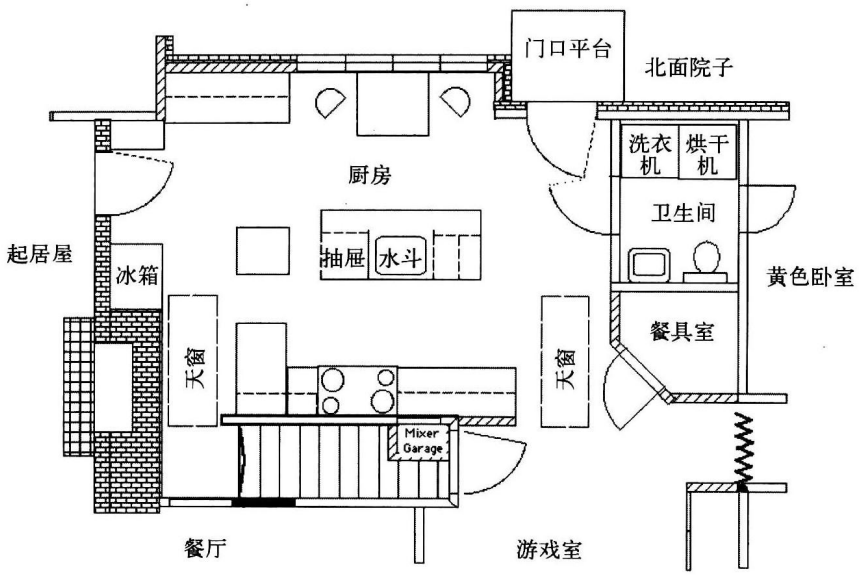


图23-2　重新建模的厨房平面图

其他设计决定

门。我们应该在起居室和厨房之间开一扇门吗？是的，这意味着要切开8英寸的砖墙，所以代价比较大。屋子的通行方式需要这扇门。从餐厅到游戏室需要一扇门吗？不需要，这两间房间的靠墙空间更有价值。

壁橱。另一个决定就是把北面过道的那些壁橱移到哪里。我们决定将它们移到游戏室的东面墙上。

食品柜架子。我们将食品柜架子（原来在南面墙上）移到新的餐具室，它是靠着游戏室的北墙建的，占据了移走管风琴后空出来的空间。

厨房与游戏室之间的开阔构造。让餐具室的门沿对角线方向打开，这增加了视觉的开阔性。

通行。南面的通道是专为厨师留的，北面的通道留给访客和屋子里所有东西向通行的人。

橱柜。将橱柜放在水斗岛的上方是一种选择，但虚拟环境下的查看表明，这会影响这间房的视觉空间。

水斗、炉子、冰箱构成的三角形的周长。小家庭委员会（Small Homes Council）建议，连接这三个工作地点的三角形的最大周长不超过26英尺。我们最后的设计周长是24英尺。

存放碟子、杯子。对于个子较矮的使用者来说，将这些东西放在抽屉里比放在橱柜里更好。

辅助岛。26英寸×26英寸的水斗岛带有一个12英寸的掀起式扩展部分，为银器、玻璃杯和

工具提供了存储空间。它也为放入冰箱的食物提供准备的地方，并为可以作为提供自助餐服务的主岛的扩展。

较低的东面工作台。东面的工作台设计得较低，以便于较矮的使用者操作。这也使得电器库可以比较高。

电器库。这部分空间嵌入在相邻的壁橱中，穿过南面的墙。

照明

天窗。两个2英尺×4英尺的天窗处于水斗岛的两端，位于房间较暗的南部的上方。这项设计决定来自于Alexander的设计模式："每个房间的日间采光应该有两个方向，最好是有三个方向"。[1]

后门。我们将结实的后门换成了一扇玻璃门。

窗户。我们安装了新的厨房窗户，并与整个早餐区域匹配。

人工照明。7条电路可以有不同的配置，重点针对用途、气氛和通行方式。

颜色方案。灰白色有助于增强亮度，并能够突出色彩。我们保留了游戏室原来的护墙板和厨房东面的墙，为砖砌的烟囱覆盖上了预制墙板。

评估

尺寸。通过墙体外推和移走食品柜，使宽度增加了2英尺9英寸，这改变了工作空间和通行空间。移动壁橱使视觉空间增长了2英尺3英寸。厨房的面积几乎增加了54平方英尺。

通行。起居室那扇代价不菲的门极大地改变了整个屋子。几乎所有的东西向通行都会通过这扇新门。从厨房可以看到黄色卧室东面的窗户和Nancy书房的西面窗户。

亮度。天窗、玻璃、灰白色的颜色基调和灯光照明改变了房间的感觉。

游戏室的效果。由于那些橱柜，游戏室明显觉得狭窄了，但仍然足够让：

- 音乐学生做准备和存放乐器箱子；
- 音乐学生在课程中组队演奏；
- 孙辈们玩耍。

南面的门。通向南面朝外的门的通道有一些局促。

满足的其他迫切需求

- 就餐区域也很适合作为会客区域。这里也是观看喂鸟的地方。
- 早餐-烹饪区域很方便——站在一个地方，就可以使用微波炉、电煎锅、烤箱、抽屉、炉子和盘子。

- 多名厨师可以方便地一起工作。
- 新的厨房很适合团体自助餐：
  - 通行方式是从餐厅进来，从起居室离开。
  - 人们从西南操作台的抽屉里取自助餐用的银餐具。
  - 盘子、碟子和供应品存放在西南操作台的橱柜里。
- 新的餐具室容量更大、更方便。
- 外观没有受到影响。

在设计中使用图纸、CAD、模型、仿真模型和虚拟环境

我们在设计上花了许多工夫，因为

- 对厨房的满意度占了屋子整体满意度的很大一部分。
- 厨房使用很频繁。
- 这个重新建模项目在很大程度上受到原有结构和通行方式的约束，有很多设计难题。
- 设计者对设计工作量的预算是没有限制的。

设计工作后来使用了各种设计工具。

图纸和CAD。大量设计是通过草图完成的，然后考虑合理性以及与原有结构的一致性，使用Macintosh上的MiniCad建筑CAD系统。MiniCad的文件作为设计文档。

大多数CAD的工作采用1/4英寸=1英尺（1：48）的比例尺，但CAD系统和两块的显示器能够方便地在屏幕上使用1：6的比例尺处理细节。1/2英寸=1英尺和1英寸=1英尺的比例尺也常常采用。

CAD设计是分层的，这些层分别是原来的厨房、移除的结构、添加的结构，以及电器和家具。

设计日志。重要设计决定背后的理由，以及做出这些决定之前的探索过程，都及时记录在设计日志中。本书的网页上有一些经过编辑的设计日志示例页。

等距离的画图工具。我们还使用了一个厨房设计工具，它提供一组等距离的网格和一组等大的电器、橱柜、操作台的图形，缩小到正确的比例，并打印在EAP塑料（electrostatically active plastic）上。这很容易使用，速度很快，并且产生了很好的结果。主要的限制就是它提供的那一组家具和单色效果。

模型。Nancy根据1/2英寸=1英尺的图纸制作了一些简单的纸板模型，以感受3D立体效果。我们后来发现这些模型比等距图纸要丰富得多：它们让人们能够从任何角度看到厨房的内部，虽然只是缩小的模型。

仿真模型。我们使用与实物等大的仿真模型来测试最重要的设计决定。这些仿真模型是非常重要的。

将外墙外推的效果用一些床垫纸板箱来仿真，目的是预测屋子的外观。内部操作台的安排用桌子、纸板和锯木架来仿真，在另一个较大建筑的内部空间里进行。然后针对内部单元的各种空间安排来排演厨房的使用场景。这是一种非常有效的方式，它让我们知道最小可容忍的空间，以及扩大这些空间的尺寸所带来的舒适感。

这反映了我以前在一个教堂建筑委员会的经验。我们最后仿真了教堂厨房的空间。后来我们发现，这是确定这些空间的唯一令人满意的方式。

虚拟环境可视化。因为我的UNC研究团队当时正在建设一个虚拟环境实验室，Nancy和我就用它来测试我们规划的厨房设计，作为对这个实验室的测试。本章首页的插图显示了头戴式显示系统生成的一张视图，这是一名观察者在虚拟厨房中走动时看到的视图。我们的追踪技术允许观察者在15英尺×18英尺的空间内自由走动，这几乎包含了整间厨房。

在20分钟到40分钟的厨房设计会议中，在现场的幻觉十分强烈——人们忘记了虚拟环境设备，注意力都集中在厨房上。

虚拟环境的发现

- 虚拟环境会议中最重要的发现就是水斗翼侧的吊橱破坏了视觉空间，使厨房感觉又小又狭窄。所以我们重新设计，取消了这些橱柜，仍然保留了所需数量的搁架。
- 早餐-烹饪区域的一盏吊灯显得很突兀，需要换成凹陷的天花板吊顶。
- 虚拟环境下的体验肯定了为巨大的烟囱墙绘制壁画的计划。
- 硬木地板的对角线排列看起来很有效果。
- 其他发现表明虚拟环境设备和技术还需要一些改进。

学到的一般经验

1）厨房实际上是屋子里最重要的房间。大量的设计工作是值得的。

2）14年之后，我们觉得应该采用不同方式设计的只有一些小细节。这个愉快的结果有一部分原因与Linux一样，设计者就是用户，所以用例是实际的、有代表性的。

另一个巨大的因素是在设计上所花的时间和精力。像System/360架构（见第24章）一样，我们有充足的时间。在软件行业，人们希望利用大量的设计时间，与真正的用户一起测试原型。同样，我们使用了很多设计时间来测试伪原型（仿真模型和虚拟环境模型），执行了大量的用例。

我确信，大多数项目需要将总时间表的更多部分投入设计工作之中。

3）与朋友们的广泛交流和咨询产生了决定性的好想法，这其中包括基本构成。

4）全尺寸的仿真模型与使用场景结合，这是非常重要的。

5）虚拟环境技术提供了重要的信息，这些信息是平面图甚至仿真模型都不能提供的，特别是关于视觉空间和房间气氛等方面。

从实用的角度来看，虚拟环境会变得更便宜、更易于使用。仿真模型不会这样。所以关键问题不是“虚拟环境提供的价值是否超过了仿真模型”，而是“仿真模型是否提供了虚拟环境无法提供的价值”。

关于这个问题，我在设计空间方面的经验和虚拟环境实验室的科学研究成果都给出了肯定的答案。Insko发现，为虚拟环境增加能够触摸的泡沫塑料仿真模型（甚至只是作为模型图片来查看），可以极大地提高现场感。[2]那些在带有可触及的实物模型而不仅仅是在只能看到示意图像的虚拟环境里接受过训练者，在走一个真实的迷宫（蒙眼）时，不仅速度有显著地提升，而且所犯的错误也比那些在同样的虚拟环境受训但只看过示意图像者大大减少了。

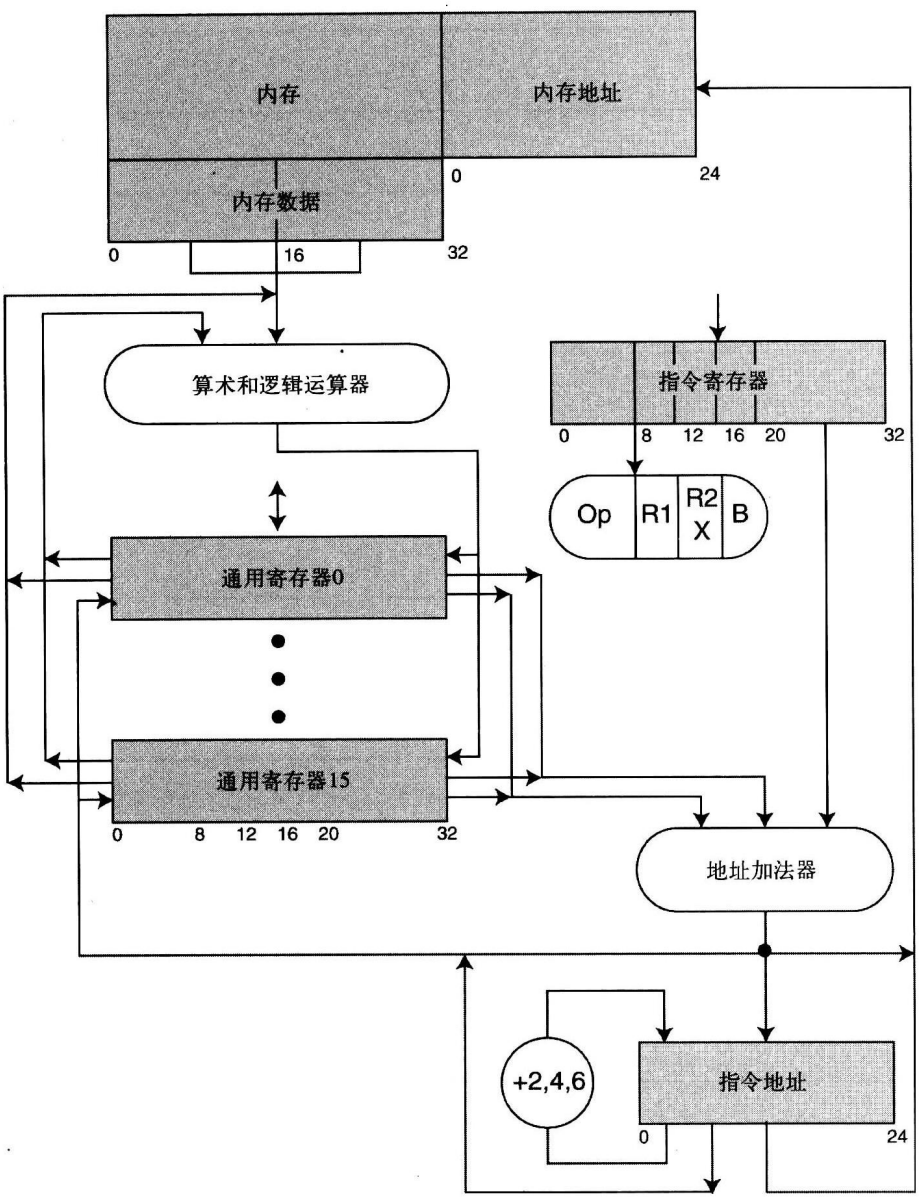
因此，我相信在设计厨房这样频繁使用的空间，或在设计大量复制的空间时，例如设计办公大楼中的办公室时，仿真模型和用户场景执行将继续有价值，值得在它们上面所花的精力和成本

注释

1. Alexander (1977)，《A Pattern Language》。

2. Insko (2001)，“Passive haptics significantly enhances virtual environments”，Whitton等(2005)，“Integrating real and virtual objects in virtual environments”。

读书笔记



IBM System/360的基本程序设计系统
Blaauw和Brooks(1997)，《Computer Architecture》，图12-78

第24章

案例研究：System/360体系结构

IBM的50亿美元豪赌。

——Tom A. Wise (1966)，《财富》杂志

（IBM的System/360大型机以及与其兼容的后续机型）……是计算机业界长久以来的主力军，而且将继续发挥作用。

——Gordon Bell (2008)

亮点和特性

最大胆的决策。舍弃IBM已有的6条产品线的所有后续研发，在一条新的产品线上孤注一掷，将已有客户群全部转让给和IBM已有产品线兼容的竞争对手的计算机产品。毋庸讳言，这种决策当然是CEO Thomas J. Watson, Jr.做出的。

大胆的决策。将新的6条计算机产品线全部规定为仅与一种体系结构严格地向上和向下二进制级兼容。这一新方案由Donald Spaulding提出，由Bob O. Evans将它形成决策。

大胆的决策。将体系结构建立在8位构成的字节的基础之上，淘汰全部已有的I/O和辅助设备，连卡片穿孔机也不例外。

项目介绍和相关背景

所有人：

IBM公司

设计师：

Gene Amdahl，架构经理；Gerrit Blaauw，次席设计师，用户手册作者；Richard Case、George Grover、William Harms、Derek Henderson、Paul Herwitz、Graham Jones、Andris Padegs、Anthony Peacock、David Reid、William Stevens、William Wright，以及Frederick

Brooks，项目经理

日期：

1961年～1964年

相关背景

很少有哪一种计算机体系结构像IBM System/360产品家族这样能够将其原理被如此彻底地加以讨论。本章“注释”一节给出了部分最重要的有关原理的讨论。[1] 因此，这篇案例研究文章只是列出其亮点所在。

在20世纪60年代，显而易见地，IBM的第二代（基于分立式晶体管技术的）计算机产品线已经在体系结构业界逐渐失宠（主要是由于其内存寻址能力太差）。IBM已有的互不兼容的、有着各自的软件和市场支持的产品线，是以下这几个：

- IBM 650（第一代，电子管）及其不兼容的基于晶体管技术的后续产品1620
- IBM 1401及其不兼容的后续产品1410
- IBM 7070-7074
- IBM 702-705-7080
- IBM 701-704-709-7090
- IBM 7030（即Stretch，仅生产了9台，市场方面再无下文）

以上头两项，以及所有列出机型的大约三分之二，由通用产品部（General Products Division，GPD）负责；其余机型由数据系统部（Data Systems Division，DSD）负责。1410和7070是直接竞争的产品，7080和7074也一样。几条产品线代表着若干个截然不同的体系结构理念和基本决策。

数据系统部业已于1959年开始研发一条新的产品线，亦即“8000系列”，它基于第二代分立式晶体管技术，反映了Stretch体系结构的设计理念，并被设计为7074、7080、7090和Stretch的替代性后续产品。首批工程样品开始试运行，8000系列的四种机型也经历了“从零开始”的成本估算、市场预测，以及定价的过程，这是1961年1月的事。市场预测的一个关键组成部分是基于电话线路进行计算机通信的一组新的应用。

1961年上半年，数据系统部内部发生了一次有关产品的暴风骤雨式的争论，问题集中在是否应该继续沿着8000系列的老路坚定不移地走下去——正如我错误地倡导的那样——还是应该花3年的时间，并基于即将到来的集成电路技术来设计一款新的产品线再投产。后一个计划，倡议者是Bob O. Evans，他最终胜出。8000系列的研发投入中止了，当年6月，一个新的、基于集成电路的数据系统部生产线开始动工。Evans遂任命我作为其负责人，这真是个完全出乎我意料的安排，它反映了Evans为人的宽宏大量。

与此同时，公司的技术顾问Donald Spaulding也逐渐接受了这个观点，即IBM需要的是一条统一的、全公司范围内的新产品线，而非仅仅是一条新的数据系统部主导的、仅仅关注高端

市场的产品线。他说服了公司副总裁T. V. Learson，后者成立了一个全公司范围内的战略委员会，即SPREAD委员会，以便研发这样一个计划。该委员被深思熟虑地置于John Haanstra的领导之下，他是通用产品部的工程副总裁，他被认为也许是在通用产品部的自治权受到任何一点儿约束时都会跳出来的最强力的反对者，而且他的1401产品线被证明取得了巨大的成功（史上首例卖出了多于一万台的计算机机型）。SPREAD委员会最终在1961年底给出了它的报告，公司管理委员会采纳了其建议的以新产品线作为全部已有产品线的继任者的提案。[2] 这个令人震惊的大胆举动后来被《财富》杂志戏称为“IBM的50亿美元豪赌”。[3] Evans称之为“把整个公司都押上去了”。我被任命为公司管控经理，以协调所有的研发活动。幸运之处在于，除了在职种上我属于拥有全公司范围的权力的人之外，面对市场需求以及整个项目在体系结构方面的生产组织时，我还是整条产品线的负责人，人事权为我提供了签署各类文件的权力，而产品线的管控权则实实在在地带来了金钱和劳动力的支持。

SPREAD的报告中要求一开始就研发6种机型，包括一种成本超低的机型和一种“超超级计算机”，并且要在两三年内就完成。前六个机型分别被命名为30、40、50、60、64以及70；后面的两种则得名20和90。20和30这两种机型由通用产品部负责研发，其余的研发责任则落在数据系统部肩上。

目标

主要目标

- 建立严格向上同时向下二进制级兼容的计算机体系结构。
- 这些机型必须在商用数据处理、科学计算和远程计算等方面适用，并提供竞争力。
- 拓展新应用的效能，在均摊到每台计算机的费用降低到原来的一半的前提下，仍然使IBM得到稳定增长的销售收入规模。我们不能依赖于IBM已有的应用所占有的市场份额有爆发式的增长，或是那些应用的数量迅速翻一番。
- 使每种机型在其各自的市场内实现（有竞争力的）性价比期望，从最便宜的机型到最快的超级计算机都是如此。

其他重要目标

- 研发单一的、新的全面软件支持，利用二进制级兼容性这一点来使用单一的、丰富的系统来替换已有的不完整的第二代软件系统。这就必须包括一个新的操作系统，以配合从第二代计算机操作系统的经验中衍生出来并迅速演变的各种概念。
- 想方设法地帮助客户从他们的第二代系统转换到System/360，即便竞争对手已经提供了IBM业已停产的产品线的后续机型。
- 提供这样一种体系结构，有时可以采用旧有的技术实现，以期能够满足IBM联邦系统部无论是军用还是政府的使用（例如美国国家航空航天局）产品之需。
- 达到可靠性和可维护性的新水准，包括极端可靠的多处理器系统。

机遇（截至1961年6月）

一种新体系结构的必要性。磁芯内存已经被证明是相当可靠的，并且已经跌到了白菜价。其后果是，所有的客户都想要更大的内存。由于全部已有的产品线都已经耗尽了它们的寻址能力，采取一个或多个体系结构方面的重大修正变得必不可少了。这给了我们一个机会来改正从第一代和第二代计算机的用例和用户那里得来的许多教训。而这些教训在旧有的体系结构中基本上是不可能加以改正的。

新的、更廉价的技术。IBM的技术部门当时正在热烈追捧集成电路，并即将达到一个重要的中转站，称为固态逻辑技术（Solid Logic Technology，SLT），而且预备在1964开始量产。这将保证对于任意给定复杂性的机型，都可以把成本降到原来的一半，并同时提供更小的尺寸、更低的能耗，以及更高的可靠性。这种性价比的激增保证了给予客户以足够的激励，去克服十分劳民伤财的、向新的不兼容系统的转换过程中的种种困难。

充裕的设计时间。新技术的来临时机意味着系统架构师有了一次难得的机会获得充裕的时间——差不多有两年——来完成深入细致的工作。

新型的I/O设备。随机存取磁盘技术进展迅速，使得全新的数据处理方法和一种从根本上不同的操作系统实现方法成为可能。

新的远程计算能力。计算机通信技术，这种最初为防空事业研发出来的技术，已开始彰显在商业应用方面的吸引力，并已率先应用于航空订票系统。

挑战和限制

兼容性——地址尺寸。到目前为止，最大的技术挑战是实现严格的（二进制级的）向上和向下兼容性，同时使各个级别上的机型在各自的市场中与神枪手般的竞争对手短兵相接。如何为最小的机型保持低成本的同时，又不会同时给超级计算机套上过分的限制？又如何让超级计算机快如闪电，又不会同时让低成本机型不堪重负？关键问题在于地址尺寸。顶级产品线需要大量的地址位数；那么，最落后的产品线（串行方式实现）能够承受内存地址的开销，以及大量取得闲置地址字节带来的性能冲击吗？

兼容性——操作指令集。如果在提供复杂的操作——例如为科学计算应用而进行的浮点运算，以及为商用数据处理而进行的字符串操作——的同时，又不能在我们对机器的成本目标方面大打折扣？

更广泛的应用范围。第三个重大的挑战是实现整体系统的多样化需求，包括新型的应用（尤其是通信系统和远程终端）、运算密集型的系统，以及数据处理密集型的系统。

从已有的系统转换。从第二代系统转换简直是一场噩梦，这是我们在做第一年的设计时未能充分投入精力去考虑的。

最重大的设计决策

由8位构成的字节。字节由8位构成，而不是像主导了第一代和第二代计算机（Stretch除外）的情形那样，字节由6位构成。这是最大的、经过了最激烈的辩论后形成的决策。它会带来诸多分歧：浮点精度主张由48位构成一个字，而由96位构成一个双字，因此一个字节只能由6位构成。一条指令若由24位构成就太短了，而若由48位构成又太长了。小写字母表又会提出怎样的表达位数用量的需求呢？这在早期的计算机上几乎是完全未知的。

对于小写字母表的未来应用确定以后，我基本上心里有数了。我们最终确定了字节由8位构成，数据字和单地址指令由32位构成，浮点字由32位和64位构成。

落败的栈式体系结构。作为地址长度问题的尝试解决方法，我们从一种栈式体系结构入手。深入了差不多6个月之后，我们发现它对于中等和高端机型运作良好，但对于低端机型则性能太差，因为在那些机型上这样一个栈只能在主存中，而不能在寄存器中实现。

设计方案竞赛。栈式体系结构落败之后，Amdahl提议我们开展一个内部的设计方案竞赛。他的想法效果不错——Amdahl的小组和Blaauw的小组各自独立地提出了采用基址寄存器的办法来解决内存尺寸问题的思路。因此，我们就采纳了它。

24位地址。我们很不情愿地将地址尺寸定在了这个数上，但规定了按字节寻址。我们明确地知道，并且我在1965年就公开地预测道这个体系结构在某个时间点必然会调整到32位，可是在1964年的实现中这是我们负担不起的。[4] 本来为了未来的发展形成了许多明智的规定，但遗憾的是，转移和链接子例程的调用指令被粗心地设计为使用了地址中的高8位，而本来这些位是不应该被触及的。

这是一个揭示团队设计风险的明显例证。我未能向整个团队成功地灌输我们关于未来扩展的观念，而且没有一次设计复查找出这个错误。

标准的I/O接口。为了实现广泛多样化的专用应用系统，我们为连接到所有I/O设备的附件设计了一个标准的逻辑的、电气的和机械的接口，就像Buchholz一开始为Stretch所做的那样。这从根本上降低了配置和软件的成本，也大大简化了I/O设备和控制单元的工程研发。

监管机制。一套经过深思熟虑监管设施被设计了出来，因此，这些系统可以由操作系统控制而无须人工干预。这包括一个中断系统、内存保护、一个特权指令模式以及一个定时器。

单个错误检测。完整的端到端的单个错误检测机制在所有的S/360实现中都是强制实施的，尽管并无证据显示客户有意为此付出代价。这大大有助于实现坚如磐石的可靠性，以实现可维护性目标。

用于商用数据处理的来自所有制造商的计算机，从最初的UNIVAC机型开始，就都介入了对该特性的密集检查。而用于科研的计算机，从最初的Burks、Goldstine和冯·诺伊曼的论文中所诞生的机型开始，却没有这样做。这似乎有点本末倒置：毕竟在原子弹爆炸时的计算中发生了一个硬件错误，要比同样的错误发生在一份公用事业账单计算时来得严重得多。我想，

产生这种差异的主要原因是科学界已经在其项目中常规地包含参与诸如能源节约的各种全面检查了。

我们观察到，在1961年，人们普遍地对他们的计算机给出的结果深信不疑，所以作为一种专业责任我们加入了硬件检查机制，并希望由此带来的额外费用不会窒息市场。

十进制算术。为了简化庞大的数据处理市场中的平台转换和用户培训的过程，我们决定加入十进制，就像原来的二进制算术一样。(和早期的650、1401、1410、7070、7074以及7080系统不同，新体系结构中所有的地址编码都采用二进制。)

提供十进制数据类型的做法也许是个错误：我们应该注意到，COBOL以及其他语言处理这个问题的方式是将货币金额存储为整型，从而就不会产生分数转换带来的误差。省略了十进制数据类型究竟会在多大程度上伤害到市场营销结果，人们只能臆测。硬件成本并不是最大的部分，软件成本以及附加的概念复杂性才是。

多进程。为多个处理器设置了规定，将它们配置到单一的系统中去，同时系统管程会将它们放到任意一个未停止工作的处理器上运行。

以微程序实现。在SPREAD报告中，我们规定，体系结构必须采用微程序实现，除非某个特定的工程经理能够指出传统逻辑组件可以提供比微程序实现高出33%的性价比优势。这使得低端处理器也可以包含相当丰富的统一操作集，唯一的成本不过是多了一丁点儿的管控用内存。60和64机型一开始的研发采用的是传统逻辑组件，然后在研发过程中改换成了单一的65机型，后者采用微程序实现。75和91机型使用的是传统逻辑组件。

模拟早期体系结构。Stewart Tucker发现，32位带4位奇偶校验位的内存和数据通道字在65机型上的实现，可以优雅地适应7090机型提供的36位不带奇偶校验位的字实现。他撰写了一个基于微代码的7090模拟器，其中十分有效地利用了65机型的数据通道。这种突破性发明被证明是一种针对7090、7074和7080客户的平台转换问题的主要解决方案。[5]

在1964年1月的一个至关重要的时间节点上，William Harms、Gerald Ottoway以及William Wright绞尽脑汁，几乎彻底未眠，在30机型上做出了一个模拟1401机型的微程序实现。这卓越地解决了最大单一客户的平台转换问题。

没有虚拟内存。在S/360的体系结构定义阶段，虚拟内存最初在剑桥大学的Altas计算机上发明出来，使用它的操作系统则是由剑桥、麻省理工学院和密歇根大学共同研发。我们就是否在设计中加入它的问题进行了长时间的艰苦辩论。最后，由于性能原因，决定不加入。这是个错误，在它的第一个后续产品System/370中就得到了纠正。

新的随机存取I/O设备。项目中在新型磁鼓上投入了大量研发精力，以备操作系统存储，还研发了新型的磁盘文件系统。我们视之为新应用和系统配置多样性的基础。相似地，新型的单线和多线通信控制器也被研发出来。

输入输出通道。I/O由相互独立的操作通道处理，本质上是专门的存储程序单元，其中有些为快速块传输而优化，另一些则支持最多256路通信的多路信道。

里程碑事件

1961年夏。新产品线的体系结构工作在数据系统部启动。来自IBM研究院的Amdahl、Boehm和Cocke加入了Blaauw来自8000系列的体系结构组。工作开始于栈式结构的尝试。

1962年1月。全公司范围的资源被组织起来了。

1962年春。首次性能评估显示，栈式结构实现不具竞争力。通过设计竞争，最终导向了基址寄存器寻址的方案。

1962年夏。字节尺寸之争尘埃落定。

1962年秋。体系结构的首个草案出台。

1963年秋。体系结构手册定稿。

1964年1月。出现了在S/360的样机和通用产品部的1401S机型之间进行的主要产品之争，后者是由Haanstra推动研制的，比1401快6倍的机型——Haanstra现在已经执掌了通用产品部总裁之职。S/360由于发明了在30机型上的1401模拟器而最终胜出。

1964年4月。30、40、50、65和75等诸机型已经发布，同时暗示90机型也即将问世。

1965年2月。首台S/360出货（40机型）。

1972年8月。带有虚拟内存的System/370发布。

1980年初。基于31位架构的System/370 XA发布。

2000年。基于64位架构的Z系列发布。[6]

结果评估

稳定性

对计算机体系结构而言，稳定性的一种定义应该是“持久性”。我预言，该体系结构将以多种实现的形式持续25年，并将会有一些修改以提供更大的地址空间。[6] 如今，离S/360发布已经过去整整45年了，这个体系结构仍然持久傲立，只是进行了迭代式的增长。最近的一个实现是IBM Z/90，于2007年3月发布。它仍然向后兼容，S/360的程序将仍然可以运行无阻。这些所谓的大型机持续不断地运行着占相当比重的全球数据库方面的工作，在它们上面运行着VMS/360和VM/360的后续产品，或者越来越多地使用Linux作为它们的操作系统。

另一种稳定性的定义是“在行业内的影响”。Gordon Bell，他本人作为一个伟大的DEC计算机的架构师，最近将System/360称为史上最有影响力的计算机体系结构，这主要是指在学术界的影响，而不是指市场占有率，因为就后者而言PC可以易如反掌地取胜。[7] S/360完成了由8位构成的字节的改变，这给计算机体系结构带来了彻底且永久的变化。它再三强调的基于磁盘的输入输出配置，也同样带来了系统设计上的根本改变。[8]

Gene Amdahl取得了S/360的许可，并在他的Amdahl公司生产的高度成功的系列机型上精确地给出了它的实现。美国无线电公司取得了该体系结构的许可，并将其用于自己的Spectra 70系列计算机。尽管美国无线电公司忠实地给出了该体系结构的问题模式的所有实现，但是它的架构师还是选择做出了一个独家的管控模式体系结构。美国无线电公司的版本被授权于、并被密集地应用于西门子、富士和日立公司。

S/360架构明显地影响了DEC的VAX系列机型，以及PDP-11系列计算机，还有它们为数甚众的微程序计算机后续产品，如摩托罗拉的6800和68000机型。

有用性——竞争力，各个市场的分析

从商业角度而言，对System/360的押宝是大大地赢了。IBM的年报显示，从1964年～1968年平均年收入涨幅是21%，同期平均利润涨幅达20%。

1964年4月7日发布了大约144款新产品。其中许多产品都提供了多种内存选项。最惊艳是一个用8位表示的I/O设备阵列：多台打印机，有些还带有可变字符集；多个磁盘，有些还带有可拆换的磁头；一系列通信终端和网络设备；新式的卡片穿孔机、读卡机和卡片打印机；其他的杂项设备如支票分拣机以及工厂数据输入终端等。这些由天各一方的实验室研发出来的极大丰富的设备集，促成了系统配备几乎无穷无尽的变种和规模。I/O接口的标准化，以及对它的软件支持意味着配置的扩张和变更十分容易。CPU的兼容性则意味着位于配置的中心位置的计算机经常过了一个周末就被升级到了一种新的机型，但是无论是I/O配置还是它的支撑软件都无须改变。

所有的机型在其各自的市场都表现优秀。30机型以及它所带有的磁盘和打印机，取得了立即的胜利。而向上兼容的20机型在不久之后投放，表现亦十分出众。

65机型在那些原本主要在7090、7094、7094II、7080、7074，以及其他机型上运行的应用方面取得了非凡的成功。在这个机型上，新的数据库技术运行得很流畅，并且它和它的后续机型在这个领域中独领风骚。它在工程计算方面也是一把好手。那些值得一提的竞争对手通常说自己的机型也是采用S/360架构的，而所谓的插接兼容机型，则通常运行着OS/360操作系统。

75和91机型，以及这个系列的其他机型是为科学计算而设计的超级计算机。它们在这个市场中，与同时代的CDC和Cray超级计算机基本上各占半壁江山，但是Cray的后续机型最终在此夺得了先机。四台75机型的超级计算机为阿波罗程序提供了陆基计算支持；而基于System/360的加固衍生机型则担当了登陆用计算机的重任。

闪光点

原始的体系结构相当洗练，仔细地做了体系结构、实现和具体技术手段的概念解耦。[9] 严格的向上和向下兼容性要求建立了严格的纪律，从而保障了低端机型不会有功能短缺，同时高端机型又不会过分臃肿。（相似地，每个作家都明白，严格的页数限制往往会导致比较言简意赅的作品。）Blaauw在操作码表中留出了恰到好处的空间，以备未来扩展之用。当然，这种扩

展肯定是有的，并且扩展以后的结果是，操作码表就不像原来那么井然有序了。

我们最大的技术失误在于没有从一开始就纳入虚拟内存技术。这是专家级设计师犯了方向性错误的一个例证（见第14章）。

我们最大的美学和概念失误在于我们未能认识到一个I/O通道其实就是一台计算机。首次在CDC 6600上亮相的Cray超级计算机的外围处理器，是一种极其优雅而强力的概念的化身。许多并发的I/O流中的每一个都由体系结构上分解出来的一台简单而小型的二进制计算机所控制，全部由同一个共时数据流实现。

在CPU体系结构中最丑陋的设计是SS指令的格式，它提供了一个基址寄存器，却未像所有其他格式一样提供一个单独的指数寄存器。正如以上已经指出的那样，子程序Branch和Link使用了高位地址，而后者本来是为扩展到32位地址而保留的。取址指令也同样不应该清除这些高位状态。

一个小一些的错误是，我们从一开始就未能为浮点操作定义提供保护位。我们只得在头一批S/360计算机交付以后，去现场修正这个问题。

也许，对于我们辛勤成果最有道理的美学批评，是在奚落S/360其实是同一个招牌下面混合了三种体系结构：最基本的32位二进制计算机；64位浮点计算机，带有不同的数据流格式；以及逐字节的处理器，带有非常不同的数据流格式，甚至实现了十进制算术（见本章卷首插图、图24-1和24-2）。事实上，如果把选择器通道和多路信道通道也算上，实际上一共有了五种体系结构。采用微代码实现使得所有这些架构并行不悖。

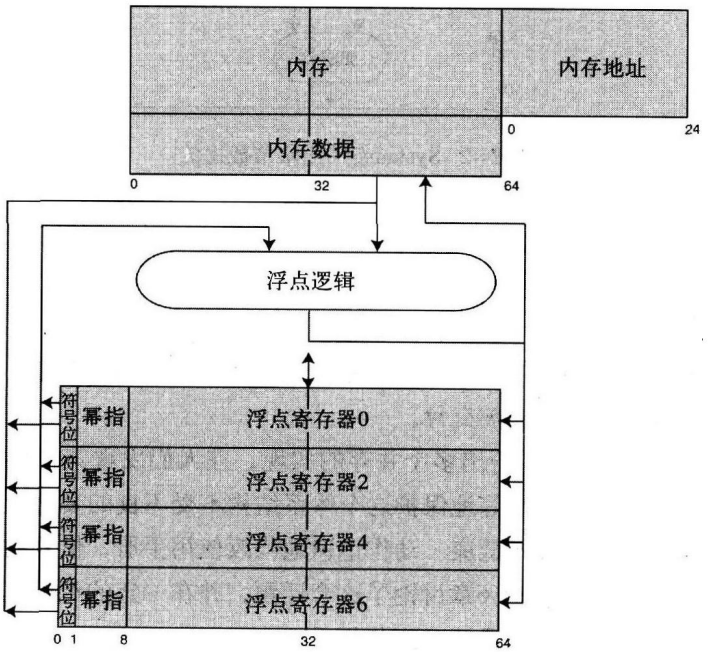


图24-1　System/360的浮点数据流

Blaauw和Brooks (1997)，《Computer Architecture》，图12-79

多个并行的体系结构的存在，达到的目的是实现了真正的通用计算机的机型系列，可以搭配适当的处理器、内存，尤其是I/O配置，从而满足各类应用和性能之需。

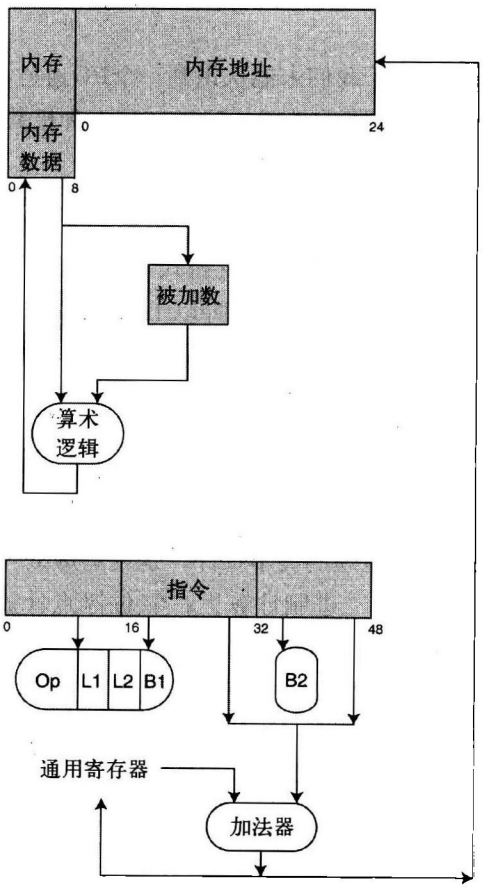


图24-2　System/360逐字节数据流

Blaauw和Brooks (1997)，《Computer Architecture》，图12-80

取得的经验教训

1）在项目安排中留出充裕的设计时间。它改善了产品质量，使其更长久地可用，并且由于减少了返工，往往还能使项目提前交付。

2）从相同的体系结构出发，给出多个并存的实现，在人们发现某种实现已经（通常是无意地）偏离了体系结构时，能够很好地保护这个体系结构不受不良的妥协之害。若是只有一种实现，往往可以更易如反掌、成本低廉、动作迅速地修改使用手册，但真的改正机型上的缺陷可就是另一回事了。《人月神话》第6章讨论了这个话题，并在一定的细节程度上提供了其他的解决方案以保证实现符合体系结构设计（而非相反）。

3）Amdahl在我们的第一个设计方案遭遇搁浅之时提出来一场设计竞赛的方式，被证明是产出颇丰的。它为很多纠纷达成了高度的共识，也迅速地聚焦了关键的差异所在。更有甚者，

它为鼓舞团队士气起到了强大的积极作用。2008年，我四十年来第一次听到Doug Baird提到此事，他当时还是团队中的新手架构师之一。他还颇为赞赏地回忆道，当时他和一些刚入行的同事能有这么一个机会把他们完成的设计拿出来和团队中其他的资深架构师分庭抗礼。

4）对于全新的设计，而非后续产品而言，从一开始就要将设计工作的一部分投入在性能及其他必要属性指标的建立上，并做好成本代理估算（比如在进行第三代计算机的寄存器位数估算时所做的）。

5）市场预测的方法论是为后续产品，而不是为全新的创新产品所设计的（违反了这一点的一个教训可以参见第19章）。全新产品的设计师们应该投入很多早期阶段的精力，以让市场预测专家们把所涉及的全新概念熟悉起来。

注释

1. 关于S/360体系结构原理的最重要论述如下：

- Amdahl (1964)，“IBM System/360体系结构”。
- Blaauw and Brooks (1964)，“System/360逻辑结构概述”。
- Blaauw and Brooks (1997)，《Computer Architecture》，12.4节。
- Evans (1986)，“System/360：回顾视角”。
- IBM公司(1961)，“处理器产品线——SPREAD任务组最终报告，1961年12月28日”。
- IBM公司(1964创刊)，《IBM System/360 Principles of Operation, Form A22-6821-0》。
- 《IBM Systems Journal》第3卷，第2期（整刊）。
- Pugh (1991)，《IBM's 360 and Early 370 Systems》。

2. IBM公司(1961)，“处理器产品线——SPREAD任务组最终报告，1961年12月28日”。

3. Wise，“IBM的50亿美元豪赌”。

4. Brooks (1965)，“计算机体系结构的未来”。

5. Tucker (1965)，“大型系统仿真”；在此事件中，他并未将7090的浮点字逐字地映射成S/360上的字，而是扩展到各个组成部分的范围来模拟。

6. http://en.wikipedia.org/wiki/System_370，此页面包含了一段关于基础体系结构的演变和要点的经典论述（最近于2008年12月访问）。http://www.answers.com/topic/ibm-system-360，该页亦如此，参见2009年8月归档。

7. Bell (2008)，“IT老军医Gorden Bell谈史上最有影响力的计算机”。

8. Bell和Newell (1971)，《Computer Structures》，第3节，561～637，给出了另一个评估结果，以及一段相当详细的讨论。

9. Blaauw和Brooks (1997)，《Computer Architecture》，在1.1节中有将这三者加以区分的详尽阐述。

一个大油桃和一串小樱桃。这是一种隐喻，象征着OS/360的大型控制程序，以及若干使得OS/360的支持程序包变得完备的较小型的、彼此独立的语言编译器和应用程序集之间的关系

来自P. Desgrieux/photocuisinet/Corbis公司

第25章

案例研究：IBM Operating System/360操作系统

软件过程中的核心矛盾来自于以下事实，那就是我们必须从发现非正式的、“存在于现实世界”的需求出发，最终却要达成一个正式的、完成“计算机内部操作”的模型。

——Bruce Blum (1996)，《Beyond Programming》

亮点和特性[1]

大胆的决策。研发单一软件包：单一的操作系统及其配套的一组编译器和实用工具程序，以供所有计算机和I/O配置使用。它能够生成适当配置以适应和利用各种内存尺寸和I/O配置。

大胆的决策。强制要求为操作系统的存储准备一个随机访问设备。

大胆的决策。不要求必须有操作员。将操作系统设计为能够使计算机系统无须人工输入或干预就可以运行。操作员仅作为计算机的手力和脚力，完成磁盘、磁带、卡片坞和打印纸的装卸。作为另一种选择，同一个操作系统也可以被配置成完全由人类操作员控制的形态。

大胆的决策。将多任务纳入可能会以并发安全模式执行，但并非专门为并发执行而设计作业和程序。

设备无关的I/O。程序只能被称为访问方法的抽象I/O数据类型编写。I/O设备类型、专用设备，以及它们所占用的空间将在作业被分配执行的时刻才落实。举例来说，同样一个归并排序程序，在一次执行中可能是从磁盘到磁盘的，在另一次执行中可能就成了从磁带到磁带的。无论输出结果是要打印还是存储，可以在运行时刻很容易改变，而程序则无须改动。

工业强度。OS/360是一个具备工业强度的操作系统，被设计为24×7运行，在崩溃时能够自动留下日志并重新启动。在它更新换代的过程中，这个特性得到了加强，所以它的后续产品仍然在24×7运行的数据库系统中被广泛使用。

远程信息处理。该系统有力地为实时数据库存取以及批处理作业执行提供远程访问支持。

原生的分时系统。该系统并非设计用于在交互式终端上进行编程和调试，所以它使用效率较低的方式来支持其运作。

虚拟内存在晚些时候加入。在原始交付的版本中，S/360计算机和OS/360软件包并未提供虚拟内存。但这种情形在第一次后续迭代中就改变了，所有版本都在1970年有了虚拟内存。

汇编语言与高级语言。尽管在1961年，也许大部分计算机程序都是使用高级语言完成的，比如Fortran、COBOL，以及Report Program Generator，汇编思维仍然影响着OS/360的部分设计。一个强大的宏汇编编译器被列为语言包之一，它代表着与科学计算和商用数据处理业界完全不同的宏应用传统思维。1966年，对一些大批量安装群集的度量显示，使用汇编编写的应用程序只占约百分之一的计算机运行时间。

项目介绍和相关背景

从1961年IBM System/360项目成立直至1965年年中，我得到了一生唯一一次的机会来管理该项目——首先是硬件，接着是Operation System/360软件包。这批系列机型于1964年4月7日发布，并于1965年2月首次出货，“第三代”计算机将是名副其实，并将（半）集成电路技术引入了计算机现货产品。

正如第一代操作系统是为第二代计算机而研发的，OS/360是首先作为第二代软件支持包而存在并为第三代计算机而研发。一体化操作系统史上鲜有先例。

System/360严格的二进制兼容性使得我们能够设计单一的支持软件包，从而能够支持整个产品系列，并且能够将研发成本分摊到所有产品线中去，而且能够在很大限度上符合与全线产品相结合的市场预期。这反过来又使得建立一个具备前所未有的丰富性和完整性的支持软件包成为可能。我是用过去时态来描述OS/360软件包的，因为它的直接后续产品仍然是大型机世界中的主力军。

术语Operating System/360或OS/360存在二义性，它们既用来描述广义上的整个支持软件包——操作系统本身、语言编译器，以及支持工具——也用于更狭义的场合，仅指操作系统本身。如卷首插图所示，我们的团队有时会把整个软件包想象成一个大桃子和许多较小的、分散的樱桃。我多数情况下使用这个术语时，仅指操作系统本身。

除了OS/360支持软件包以外，最初还计划并交付了一个基础磁带支持软件包，包括一个Fortran语言编译器，这为一个无磁盘的小内存系统提供支持；另外还有一个基础穿孔卡片支持软件包。OS/360支持软件包的本意是针对所有配备16K以上内存的系统研发的。我们在如此捉襟见肘的内存中连最起码的功能都实现不了，所以我们把最小内存的需求提升到了64K。对于小型系统客户的关怀，以及OS/360的延迟发布导致了公司启动了一个完全独立的、为较小的内存尺寸优化过的支持软件包，即后来众所周知的Disk Operating System/360或DOS/360。[2] 同样的，它也在不断演化，并且它的后续产品今天还在使用。

System/360系列机型

第24章描述了计算机机型系列的市场格局，以及其主要的体系结构分野。本质的概念创新在于所有的机型（除了最廉价的一款20机型）在逻辑上完全一致，都是与单一的体系结构向上并且向下兼容的实现。Blaauw和我定义了一种计算机体系结构来使得一组计算机属性精确地对程序员可见，但不包括速度。[3] 用软件工程的术语来说，我们的狭义概念中的计算机体系结构和一种抽象数据类型是等价的。它定义了有效数据集、它们的抽象表示，以及应用于这些数据集的操作集本身的语法和语义。那么，每种机型的实现就是这种类型的一种接口。所有仿真器和所有模拟器从概念上讲也都是这样。在实践中，我们的首批硬件实现的数据流宽度范围方面包括从8位到64位，还支持多种内存和线路响应速度。

1961年的软件格局

操作系统。第一代操作系统产生了明显的分野：它们要么是被设计用于为科学计算，要么就是用于商业数据处理。它们都是批处理操作系统，设计出来是为了控制独立作业流的序列化处理。

无论是哪种操作系统都由三种组件组成，这些组件彼此独立演化。监督进程，它常驻内存，是由早期的中断处理例程发展而来的。而数据管理组件由已经演化成为一个I/O例程的标准库而链入应用程序。调度进程，它通常存储在磁带上，在两个作业之间被装入内存，以指定装载的磁带（和卡片）文件，以及产生输出的位置偏移量。操作系统提供了磁盘文件，但它本身一般是存储在磁带上的。

较新的第一代IBM操作系统提供了假脱机（Simultaneous Peripheral Operation On Line, SPOOL），所以在任意指定时刻，一台第二代计算机都可以执行一道主应用，以及若干个卡片到磁带/磁盘、磁带/磁盘到卡片，以及磁带/磁盘到打印机的实用工具。后者属于“信任程序”，被仔细地编写以避免对通常运行在磁带到磁带或磁盘到磁盘模式下的主应用构成破坏或侵入。综上，一台计算机可以一边为下一道作业准备磁带，一边运行主作业，还一边进行上一道作业的打印输出，所有这些都是同时发生的。

语言编译器。IBM的客户使用种类繁多的高级语言，而IBM致力于为这些语言提供编译器。用户最多的语言是Fortran和COBOL。ALGOL则在欧洲比较流行。而在低端系列中，Report Program Generator（RPG）则在那些从穿孔卡片机转换而来的机型中较受青睐。

而汇编语言编译器的演化，从技术角度来看很有意思。经典的两次扫描型汇编编译器已经发展为两次预扫描，后者被视为一个宏操作生成器，具备丰富的编译期功能，包括分支和循环。所以，这种宏汇编编译器以两种相当不同的方式被使用。

科学计算业界通常使用程序员自定义的宏作为开放式例程，以便完成频繁使用的操作，如矩阵计算。这些宏不仅带来了编码的简化，还加快了运行期的速度，避免了例程调用的开销。

与此形成对照的是，许多商用数据处理机构都发展了这样的实践：他们有一个小团体的编

程高手来写出一堆“内部”宏库，用以定义新的数据类型，并附带一些数据结构和操作来定义一种专门化的程序设计语言以供该机构的商业实务之用。更大范围内的程序员仅仅利用这个库里的宏，一般不会创建任何新的自定义宏。

实用工具。各种各样的实用工具，默默无闻却不可或缺地使得每个计算机软件包变得完整：排序程序生成器、媒体转换器、格式转换器、调试助手，以及崩溃处理程序。

免费软件。在当时，制造商们为了刺激硬件的销售和应用，会分发操作系统和编译器。因此，软件包的成本就必须包含在硬件价格里。[4]

接受挑战

在一个全新的软件支持包上开展工作带来机遇，也会带来很多挑战，因为这需要决定哪些是放到“下一步”才在软件支持中实现的。有些挑战被接受了，另一些则被拒绝了。

普适性。尽管上一代软件支持包在应用程序领域和性能级别上相差甚远，但是OS/360软件包却是被设计为覆盖全范围的应用的。System/360，顾名思义，就是指“包罗万象的计算机系统”。它被设计出来也是为了覆盖一个极大的性能范围，从相当节俭的64K内存的系统到最精密复杂的超级计算机系统，或是海量的数据库配置。

对这一挑战的回应给编程语言及其编译器带来了巨大的影响。一种新型的通用编程语言——PL/I，在IBM用户社区的科学和商业用户的通力合作之下研发出来。针对不同内存尺寸而设计的多种编译器其构建在多种语言上：Fortran、COBOL、汇编以及PL/I。负责编译器和实用工具的项目组会将创意拿到用户社区中做评估，对每一种创意都会纳入如何比上一代产品做出有所超越的考量。在此，我只谈最富创新的组件，亦即操作系统本身。

磁盘驻留。一种廉价的磁盘驱动器变得可用，即IBM 2311，它具备当时看来巨大的容量7MB，这意味着可以在设计操作系统时假定它驻留在“随机存取”设备而非磁带上。这是在设计概念中发生最大变化的一个。操作系统中的特定模块可以根据需要迅速地调入内存，而且各个模块可以只有很小体积、并仅完成特定功能。

一种新的、按字并行的磁鼓提供了延迟低、数据传输速率高的操作系统存储介质，以备更高性能的计算机系统使用。

多道程序。OS/360实现了一个巨大飞跃，允许多个独立的、彼此无信任关系的程序执行并行操作——这种飞跃之所以可能实现，是由于System/360体系结构提供了的硬件监管机制。早期的OS/360版本仅支持多个固定尺寸的任务，这样一来内存分配的方式直截了当。仅仅两年之内，其MVS版本就支持完全一般化的多道程序了。事实证明，实现这一点的难度远高于我们的预期。

由操作系统而非操作员来控制。一个核心新概念是，现在的常规做法是由操作系统而非操作员来控制计算机。晚至1987年，有一些超级计算机，如CDC公司的衍生产品线ETA的机型

ETA 10，仍然在操作员手动控制下运行。由操作系统控制的必然结果——率先由Stretch机型实现，在今天已是例行公事——就是键盘或控制台也成为另一种普通的I/O设备，上面只有很少几个按钮用以直接完成任何操作（例如开关、重启等）。

远程处理，但并非分时机制。OS/360从最底层的设计开始就是一个用于远程处理的系统，但它并非基于终端的分时系统。这一概念与同时代的麻省理工学院的Multics系统相悖。OS/360是为工业强度的各种规模的科学和数据处理应用而设计的，而Multics是作为一个探索性的系统而被设计出来，主要用于程序研发的。

24×7健壮地运行。OS/360旨在提供“核查并重启”机制的核查点，以感知硬件错误，并且在无论硬件还是软件故障发生后能够重新启动。当用于多处理器配置时，诊断结果能够启用一个运行良好的处理器、搁置有问题的处理器并移交其负担的工作。从一开始，OS/360就打算要全天候可用，但这是采取了一些演化步骤以后才最终实现。

设计决策[5]

系统架构

在OS/360中，控制程序演化的三支彼此独立的分流聚合到一起了。监管进程，从早期的中断处理例程演变而来；调度进程，从早期基于磁带的作业调度器演变而来；数据管理系统，从早期的I/O例程包演变而来。该系统的架构反映了它多元化的谱系。

监管进程。由于原始的监管进程只处理程序中断，所以它只在任务之间分配处理器的指令计数即可，而多道程序下的监管进程就必须同时分配主存空间了。OS/360的监管程序在任务之间根据优先级进行内存块和时钟周期的分配。

OS/360的监管进程通过控制指令控制台来实现对计算机的控制。它在每一个瞬间只将控制权出让给一道程序。任何程序故障，包括任何违反系统保护机制的企图，都会触发一个中断，将指令控制台交还监管进程。从I/O设备发回的异步事件报告，例如操作完成信号，做的是同样的事情。此外，监管程序还控制着一个受保护的、会发出中断信号的管理流逝时间的时钟，所以它可以在任意指定的时间间隔以后夺回控制权，所以它可以中止有缺陷的程序中存在的死循环。只有监管程序可以设定多种内存并施加其他保护措施，以及执行其他的特权操作，比如I/O控制等。

当一个普通的应用程序想向监管程序申请一项服务，比如想使用更多的内存块时，它通过称为监管程序调用的硬件操作来发起一次请求。这是一种积极的中断，在指令中携带给监管程序的参数。因此，对监管程序的访问只能是限定性的，在监管程序规定的允许列表之内才能通行。

监管程序还提供使得互不了解的程序之间能够在运行时相互通信的机制。

调度进程。OS/360的调度进程先准备好并发执行的若干项彼此独立的“作业”，尔后管理

每项作业中顺序执行的“任务”，如编译、链接到库、执行、输出转换等。当一道作业已经就绪，可以进行调度的时刻，调试进程先检查作业的优先级，分配全部所需的I/O设备，给出操作员指令以装载全部离线的数据存储卷，并将作业置入队列以备执行。监管进程随后分配初始内存，并初始化首个任务。产生输出之后，调度进程完成位移，并将所有已完成的数据存储卷加以卸载。

OS/360比任何它的之前产品都更明确地将调度期视为一种绑定的场合，它与编译期相比有着更多限制，与运行期相比有着不同开销。不仅在调度期分立的编译程序模块被链接例程Linker彼此绑定到一起，而且数据集的名字也仅在调度期才被绑定到特定的数据集实体和特定的设备上去。这种绑定是使用Job Control Language指定的，并由调度进程执行。

数据管理。尽管严格的程序兼容性是System/360系列机型最鲜明的新概念，但是丰富多样的I/O设备却是它最重要的系统属性，无论是应用的广泛、配置的灵活还是性能的增强都体现了这个属性。单一的机械、电气和逻辑I/O接口从根本上降低了新I/O设备的工程费用，并从根本上简化了系统配置，也从根本上缓解了配置的扩张和变更。

至关重要的软件创新在于它补充和利用了标准硬件I/O接口，这就是标准软件接口——用于对所有类型的I/O设备进行I/O控制和数据管理的单一系统。我把这个视为在OS/360中最重要的创新。

由此产生的一个新特性称为设备无关的I/O。撰写应用程序的软件工程师只使用数据集的名字即可。它们可以绑定到特定的数据集实体，绑定到特定磁带卷轴上的磁带，从磁带到磁盘，从磁盘到通信线路或打印机，所有这些通常都推迟到调度期完成。

为利用一组新的磁盘类型，专门设计了四种访问方法，用于跨越整个磁盘应用范围。它们体现了在基于缓冲或块传输的动态灵活性和最优性能之间不同的取舍原则。

- 顺序访问方法——类磁带的、基于缓冲区的

例如：用于排序（用于磁带、打印机、卡片坞以及磁盘）

- 直接访问方法——对记录的纯粹随机访问

例如：用于航空订票系统

- 分区访问方法——快速固定块传输

例如：用于操作系统诸模块

- 带索引的顺序访问方法——顺序的、带缓冲的，但能够迅速处理随机查询

例如：用于公共事业账单处理

专门为终端和高速远程通信提供充分的灵活性和易用性，还设计了两种访问方法。

在所有的I/O设备中，只有支票分拣器特立独行，它构成了对操作系统的一个非常死板的

性能约束——纸质支票在读取头和分拣袋之间飞移所用的时间固定而且短促。在银行的支票转运和支票处理设施中，这些机器会在一个读取站中读取支票底部用磁性油墨打印的数字，尔后将支票转运到24个左右的袋子中的一个里，最大速率不超过40张支票/秒。[6]

结果评估

成功之处

功能齐全，普遍适用。OS/360为操作系统功能建立了新的基准线。它实实在在地支持了一个令人惊异的大范围内的应用程序、系统配置和不同性能的计算机。

健壮性。健壮性水准无出其右。它是具备工业强度的操作系统，并已经成为运行占用大多数大型机时间的海量数据库应用的标准系统。

数据管理系统。设备无关的I/O成为编程任务的一种主要简化途径，而且为数据中心的运营和演化提供了巨大灵活性。利用节假日对处理器和I/O设备进行重新配置成为了例行公事。而经过这种重新配置以后，绝大多数应用都无须重新编译即可运行无碍。

支持远程处理。OS/360成了为银行业、零售业及其他大多数行业提供广泛网络终端的基础。

引入了虚拟内存。当IBM在System/370的后续产品线中采纳了虚拟内存以后，OS/360就成为了OS/360多路虚拟系统（OS/360 Multiple Virtual Systems，OS/360 MVS）的基础，这是一次扩充，但不是彻底重写。

Amdahl、日立以及富士通。绝大多数S/360插接兼容机型的制造商们都没有提供自身的软件系统，而是使用了OS/360软件包。

设计中的不足

系统。OS/360太过丰富了。系统常驻磁盘的事实，取消了对于早期操作系统设计师们必须严格遵守的空间尺寸约束——我们加入了很多功能部件以期收获边际效用。[7]特性成灾直到现在还未在软件社区中销声匿迹。

它提供了两种相当不同的调试系统，一种为终端上的交互式应用而构建，可以实现快速重新编译，另一种则为批处理而构建。这可以说是史上为批处理设计的最好的调试系统，只可惜才刚问世就已过时了。

OS/360的系统生成过程极其灵活和繁复。我们本应该配置一个小数目的标准软件包来迎合大多数用户的需求，而将以上这些强大功能作为需要完全灵活的配置过程时的补充。

控制块。模块之间的通信是通过全系统范围内共享的控制块来进行的，这些控制块中的每一个都由一组构造好的变量组成，并由若干个模块存取。随便哪个软件工程师都可以访问所有的控制块。假使我们早在1963年就理解并采纳了Parnas在1971年才提出的信息隐藏策略，我们

就能得以避免在原始方案中的头疼之处，以及它们带来的所有可维护性恶果。面向对象的程序设计是信息隐藏思想在今日的体现，我们都明白它的确具有优越性。

虚拟内存。正如在第24章中所讨论的那样，我们没能赶上在最初的处理器中加入虚拟内存的时机，几年之后不得不回头把它加上。与一开始就将其纳入设计相比，这个对OS/360而言非有不可的扩展就变得更加困难，费用也更高了。

调度进程的Job Control Language。无论和谁的以及和哪里的设计相比，Job Control Language可能都是史上最糟糕的程序设计语言了——它是在我本人的管理之下设计出来的。整个概念都是错的。我们根本没有将其视为一种程序设计语言，而是视为“在作业执行之前的一些控制卡片”。我已经在第14章中阐述了它的缺陷。

数据管理系统的复杂性。我们本应该与为IBM早期磁盘建立的基于关键计数数据的可变长度块结构划清界限，并另外设计一两个尺寸的固定长度块应用于所有的随机存取设备才对。

I/O设备-控制单元-通道的连接树具有不必要的复杂性。[8]我们本应该为每种设备指定一个（也许是虚拟的）通道，并为每种设备指定一个（也许是虚拟的）控制单元。

我相信我们本来可以发明一种顺序的磁盘存取方法，并能够结合以下三者的优化之处：SAM、PAM和ISAM。

流程中的不足

我在《人月神话》中已经就这个主题发表过长篇大论。在这里，我只强调两点。

我坚信，如果我们使用PL/I——当时可用的最好的高级语言来构建所有的东西，操作系统将会一样快速，更为洗练，并且更可靠，完成时间还会大大缩短。但事实上，它是使用PLS构建的，PLS只是在汇编语言的基础上加了一层语法糖衣罢了。如果使用PL/I（或任何高级语言），那么就要求我们对工作人员进行仔细培训，以使他们明白怎样写出像样的PL/I代码以及PL/I源码库，以编译成在运行期执行速度很快的指令码。

我们本应该对所有接口维持一个相对固定的架构控制，坚持对所有的外部变量声明都必须只能为库所包含，而不是在每个实例中都包含一个新的声明。如果这样做了，那么许多缺陷本来是可以避免的。

设计师团队

整个OS/360软件包的研发工作大约投入了1 000人。在此，我只指出对于其概念结构贡献最突出的团队和个人。

关键角色

实验室：波基普西市、恩迪科特市、圣何塞市、纽约市、赫斯利镇（英国）以及拉戈代（法国）

OS/360架构师：Martin Belsky

核心人物： Bernie Witt、George Mealy以及William Clark

管控项目经理：Scott Locken

编译器和实用工具经理：Dick Case

OS/360助理项目经理：Dick Case

OS/360经理（1965年起）：Fritz Trapnell

最好的一篇文档是它的概念和设施手册，这是由Bernard Witt撰写。[9,10]

取得的经验教训

1）给予系统架构师以充分的设计授权（见第19章）。“数百万美元的失误”已经在《人月神话》第47页及之后所有篇幅中有了更充分的讨论。

2）使用必要的时间来做完善的设计和原型，无论日程安排的压力有多大。项目将提前而不是延后完成，前提是可以这样安排时间。第21～24章说明了设计时间充足的好处，这里则举了一个设计时间不足的反例。

注释

1. 本文改编自Brooks (2002)，“IBM Operating System/360的历史”，发表于Broy和Denert (2002)，《Software Pioneers》。材料主要来自《IBM Systems Journal》第5卷，第1期(1966)。

2. http://en.wikipedia.org/wiki/DOS/360_and_successors，最近于2009年8月访问。

3. Blaauw和Brooks (1997)，《Computer Architecture》，1.1节。

4. Grad (2002)，“一些个人记忆”，描述了1969年对软件和硬件的解绑。

5. Pugh (1991)，《IBM's 360 and Early 370 Systems》，给出了OS/360的早期研发阶段的详细历史。

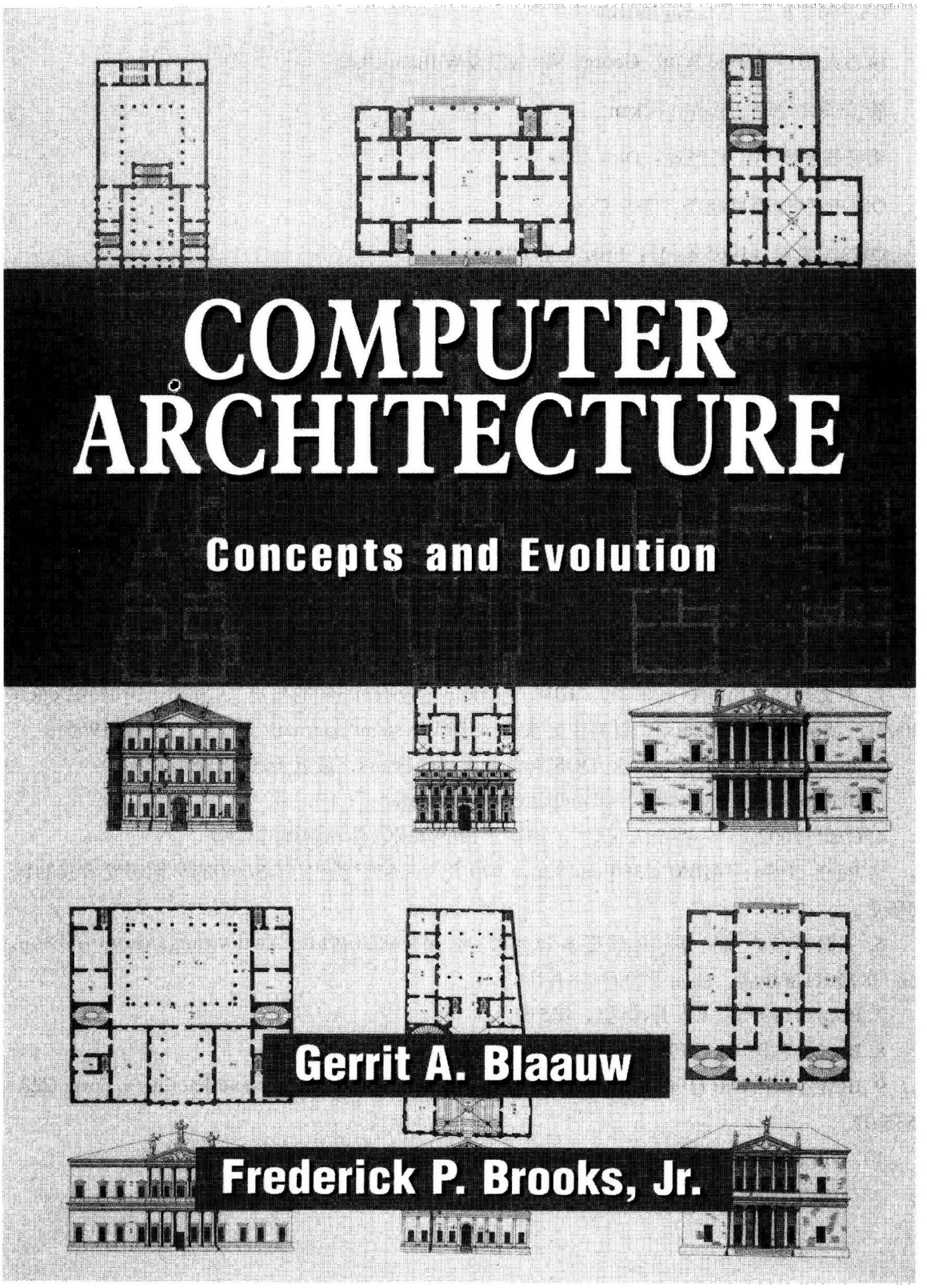
6. 一台后续产品机型样机的更多信息和一张照片可以在http://www.thegalleryofoldiron.com/3890.HTM找到，最近于2009年8月访问。

7. Brooks (1975)，《人月神话》，第5章。

8. Blaauw和Brooks (1997)，《Computer Architecture》，8.22节。

9. IBM公司和Witt (1965)，《IBM Operating System/360, Concepts and Facilities, Form C28-6535-0》。

10. Witt (1994)，《Software Architecture and Design》，阐述了设计概念和途径。



Blaauw和Brooks (1997)，《Computer Architecture》，书皮谚语

第26章

案例研究：《Computer Architecture: Concepts and Evolution》图书设计[⊖]

写书这件事是对数收敛的。

亮点和特性

大胆的决策。坚持给计算机架构一个狭义但是相当精确的定义，以此作为作品的范围。尽管我们在1962年才初次引介了这个术语，1964年它才有了一个精确的定义，之后它才在较宽泛的意义下广泛使用。我们仔细地定义并区分了架构、实现以及手段。我们只讨论架构，它的定义包含计算机的一些属性，用以管理哪些程序可以运行，以及它们的运行结果是什么，但不包括速度度量。这种精确性使得定义程序兼容性成为可能。

大胆的决策。纳入30机型的计算机架构的“动物园式”枚举，以标准化格式描述。这个格式中包括一段以散文式描述的“要点和特点”、一段对历史和技术背景的短小描述、一个图解的程序设计模型、对设计决策的列举，以及对数据表示、格式和主要操作的精确图解及APL描述。

大胆的决策。构建、测试并发布“动物园”中各种计算机架构的可执行模拟器，皆用APL撰写。“动物园”中的每种机型都包括一个可执行的APL程序，用来模拟该机型中取址、解码，以及对适当数据存取及操作例程的调用。其主要操作也同样使用可执行的APL函数描述。构建这些模拟器的过程迫使我们对这些机型实施细节的彻查，从而给这些描述增添了很大的精确性。但并无证据显示很多人使用过这些模拟器。

信息矩阵组织。构成计算机架构的设计决策分两次加以讨论：一次是系统地按照决策域顺序进行；另一次是以每种特定的机型为依据讨论与其相关的所有设计决策。

决策树。我们使用决策树作为一个表示设计选择的正式工具。大约80棵树被链接到单一的、庞大的、统一的计算机架构的决策树里。当然，这种形式主义把设计过程当做在有着良好定义

⊖ 本章中提到的本书均指《Computer Architecture》。——编辑注

的空间内进行搜索的问题解决手段，这正是我在本书中口诛笔伐的模型！我的设计观在完成那本书以后的岁月里已经得到了广度和深度的延拓。

计算机架构的演变：分野与融合。我们涵盖了早至最初的机器（Babbage机械计算机），晚至1985年的计算机架构演化过程，展示了它们在实验阶段的大相径庭，以及随后的殊途同归，最终达到了令人惊异的标准化的架构。其中汇集了许多早期机型的文档，也有很多有关现代机型的。

本书同时也是一部研究专著。我们在System/360方面，以及本书上所做的工作取得了诸多无法零碎发表的研究成果。因此，这本书包含了许多新发表的成果，并在前言中一一列举了。这算是在它看起来只是一本面向实践者和学生的教材之外的一点特别之处。

全面的参考文献。术语都经过仔细定义。丰富的主题索引专门将读者引导至那些定义及其大篇幅的讨论，以及它们所出现的地方。还有单独的人名和机型名称索引。参考书目包含超过500项的内容。本书在它作为教材用于教学之后很久，仍然是一本有用的参考书。

项目介绍和相关背景

作者：

Gerrit A. Blaauw和Frederick Brooks

日期：

约1971年～1997年

相关背景

我们两个都已经离开了计算机架构工作的一线，而在从事该科目的教学。这本书的成形要归因于我们需要教材。大约在各自经过两次课程管理之后，我们答应设计一本书，主要目的并非作为学生课本，而是作为实践者的系统化论著。我们也确实编写了大量练习，以供课堂教学及自修之用。

项目目标

摘自《Computer Architecture》的前言：

“我们想通过本书达成的目标是给出有关计算机架构之艺术的全面论述。本书主要目的不是为了作为教科书，而是作为一线架构师的参考指南，以及作为一本提出计算机架构之新概念框架的研究专著。我们提供了足够多的演化历史材料，以及读者能够不仅明白当前的实践情况，也能了解为何它们成为了现在的样子，以及有哪些做法经过试验和丢弃过的。我们的目标是展示那些不常用的另类设计思路，同时把常用的设计思路给予分析和系统化。

“看来，提供一个在计算机架构的设计过程中产生的问题的概要汇编，并给出对于设计问

题诸多已知的解决方案有哪些正反两方面的评价因素的讨论，是有用的工作。这样，每个架构师就可以根据他自己的应用、技术、品位和判断力，为这些因素施加他自己的一组权重了。”

机遇

由于我们在IBM有过8年工作上的密切合作，沟通过程很顺畅，思维模式也彼此熟悉。这使得我们的远程合作十分简单，如第7章所述。

我们曾在3种计算机架构设计上共事，而我们又分别参加过所有其他架构的设计。这些项目引导我们去研究之前机型的设计，而教学工作又巩固了我们对这些作品及其意义的认识。我们拥有的那些机型中大多数的程序设计手册。

System/360架构的设计并不是匆匆忙忙完成的，因为它所仰仗的半集成电路技术在1964年之前还没有准备好。因此，那些设计决策是经过彻底辩论的，我们也在这么一个背景下了解了很多架构设计问题中的辩证法。

为写书而做准备的时间和精力预算基本上是无限的，或者这是我们的一厢情愿。这是一个重大错误。事实上，本书的问世之际已经错过了它可以施加最大影响力和发挥最大作用的时机。

约束

我们两人都有手头在做的研究项目和课程计划，也都有需要照顾的孩子。而这本书开始写作以后，我们又都各自发表了一本相关内容的图书。因此，本书的进度就常常被我们冷落了。

设计决策

次序。对于任何说明性文字的写作来说，次序都是最难的一个设计决定。一般的关联概念图必须被修剪成树状结构，才能被映射成线性结构的文本。

我们发现了两种主要的可能顺序，每一种都很重要。所以我们两种顺序都采纳了，还加了许多交叉索引。第一部分将设计决策按概念顺序系统化地讨论。但是每种实际的决策都只能在那种机型的所有其他决策的上下文里做出。所以，我们在第二部分“机型动物园”中将设计决策放在若干机型标本的上下文里加以说明。

在本章开始的“要点和特点”一节已经列举了其他的主要设计决策，在此不再赘述。

结果评估

稳定性。如果依照持久性来判断，设计是稳定的。13年来，其有用性并未消失，其讨论也并未过时，尽管它必须由描述更新研发进展的材料加以补充。

销售情况。这本书出版得太迟了一些，它的一些潜力未能充分发挥。如果只为一两门架构

课程选择教材，人们会选择Hennessy和Patterson所著的内容精湛并持续更新的《Computer Architecture: A Quantitative Approach》替代本书。

专业的计算机架构师需要熟悉本书，既是为了了解它的前导产品如何工作，也是为了把它用作参考指南。本书有着小众但痴迷的追随者，主要来自计算机架构师。

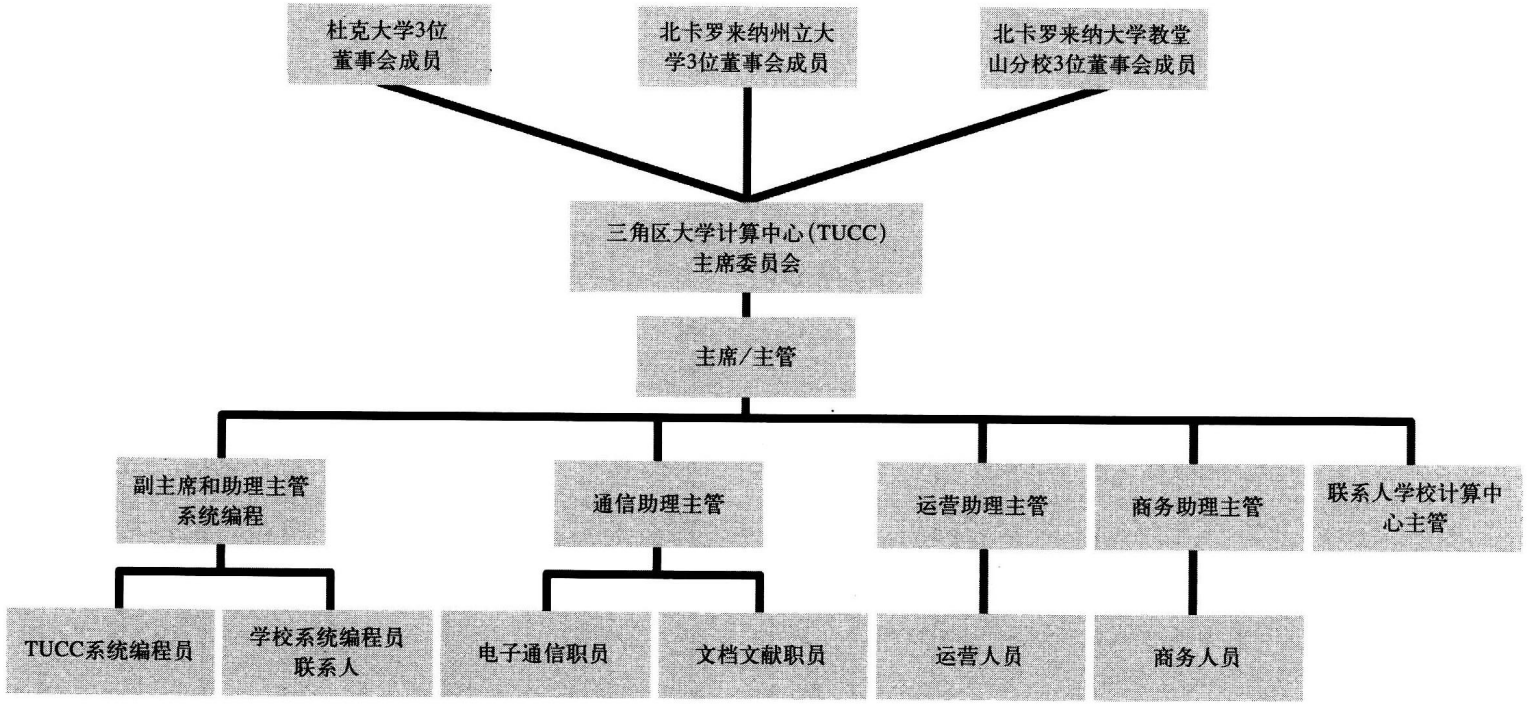
闪光点。这些就留给别人来评价吧。

经验教训

1）也许一本不那么激进却更快出版的书，对于从业者来说会更有用一些。当我身处计算机架构课程的实际教学中，我会重点讲述“动物园”和“标本”，当遇到活生生的例子时才进行设计决策的讨论，而不是系统化地照本宣科。也许我们应该把那部分独立出来写作并出版，夺得先机，而且效率更高。但这也不是一件容易的事：很多有关“动物园”的讨论都假定里面的概念都在第一部分“设计决策”中引入和展开阐述过了。

2）写书这件事是对数收敛的。核查最后一些没有把握的事实、修正最后一些有点小偏差的数字、校订最后一些不明不白的参考文献——这些工作要占到整个工作的很夸张的比例。最难缠的琐碎工作总是被延后，直到最后一刻才不得不动手处理。

读书笔记



1980年的三角区大学计算中心（TUCC）组织结构图

第27章

案例研究：联合计算中心组织：三角区大学计算中心

计算的作用是洞察，而不是数字。

——Richard W Hamming(1962)，《Numerical Methods for Scientists and Engineers》

亮点和特性

大胆的决定。建立这样的一个联合中心，三个大学将资源集中起来，共同拥有并运营一个高性能计算中心。

集中资源。为了利用二次方的性能/价格曲线，资源需要集中起来。在当时以及之后的许多年，使用n倍的时间能换来最少n^2倍时间的计算能力。这一事实为克服共同建立中心所面临的眼前和可预计的困难提供了强大的经济驱动。

无组织模型。联合的学术计算中心这一概念在当时还未出现，至少就我们所知是这样，所以没有现成可用的组织模型。

决策驱动的能源。设计中可预算的开支是能源。如何保证所有方各自的利益，同时又支持有效的决策呢？

多样的应用。有些所有方既使用中心做学术计算又使用它进行有关管理性的计算，其他的仅仅使用学术计算。

中立的地点。在三角研究园区投资了一所大楼，到每个校园都是相等的距离。

远程信息处理至关重要，同时仍然不够。所购买的IBM System/360设备是为远程作业输入和交互计算所设计的，并具备软件支持。它初始是用于远程作业输入模式，同时有着优先级系统强调小作业的快速周转。差送服务使用旅行车来回运送磁带和磁盘以提供高带宽的大数据集传输。

对全州的影响。在1964年，全北卡罗来纳州除了TUCC的所有方以外，很少有高等院校

有任何计算能力或者了解如何实施。另一个独立的组织——北卡罗来纳计算机培训计划，由北卡罗来纳州高等教育联合会领导，租用TUCC的计算能力并为全州其他大学和院校提供如下服务：

- 一年免费的计算时间，每月不超过100个任务。
- 免费一年使用安装在校园里的电报机。
- 传道者提供免费的服务，通过访问校园、培训老师、组织研讨会、提供电话咨询和疑难解决等，向高校介绍并引入计算能力。

100所以上的高校享受了这样的服务，其中许多院校持续多年投入经费继续租用TUCC。

持续性。TUCC组织历经18年，即使三个共同所有方相关的需求都各不一样，但是事实证明该方法行之有效。由于微机的出现它逐渐过时了，虽然多年以后，在另一种组织形式下，出现了多院校联合运营的专注于科学应用的超级计算中心。

介绍与内容

地址：

三角研究园区，NC

所有方：

杜克大学，私立大学（Durham, NC）
北卡罗来纳州立大学（Raleigh, NC）
北卡罗来纳大学教堂山分校

组织设计者：

TUCC董事会

日期：

1964年～1992年

内容

杜克大学、北卡罗来纳州立大学、北卡罗来纳大学教堂山分校都拥有第一代的计算机，由集中式的计算中心运营。每所学校都需要升级计算能力与容量。每所学校都需要比它们所能负担的更多的计算资源。

这三所大学决定集中资源运行一个单一的现代的高性能设施（比它们在没有帮助的情况下可共同承担的资源更多）。

国家科学基金划拨了可观的资助，其中一部分是为了资助探索提供科研学术计算服务的新型组织模型。

IBM在三角研究园区有新产品开发和加工的设施，租用了（TUCC）夜班时间，对于该项目的头3年提供了非常大的帮助 。

设计的问题是如何可良好管理地来组织这一联合设施。

目标[1]

主要目标

计算中心的主要目标。交付一个迅速的、高质量的计算服务，为客户提供多样的应用并适应广泛的复杂性。

组织设计的主要目标。为这一由3所院校出于不同需求与目标而建立并平等拥有的联合计算中心开发一个平稳运行的管理计划。

其他目标

- 保持中心的财政稳定。
- 保证决策高效且迅速。
- 确保每个所有方的用户可以公平享用所有的资源。
- 保证每个所有方的投资安全。
- 确保不会因为危险的自私自利的行为而对所有方造成损失。
- 确保中心作为一个整体运行，而不是三个部分，以实现规模经济的效益——一组职员、一系列的设备和同一个任务流。
- 支持所有方变更其对联合中心的贡献，从而相应增加或减少对应享有的服务。

机会

规模经济。在设计的时候，运行一个大型中心而不是3个小的计算设施所带来的规模经济效益是非常显著的，特别是由24小时运行的职员节省和计算机租用的节省所带来的。因为计算机、内存、磁盘以及其他的I/O设备都遵循平方律的性能/价格曲线。

可行的远程操作。由新的第三代计算机所带来的新技术第一次使得远程提交和接收计算任务以及进行远程交互会话变得可行。

国家性的原型。通过迅速的发展，TUCC可以成为地区性计算中心模型的一个先行者。一个在全美国范围内都知道的大型计算中心并且提高了北卡罗来纳州相应的新研究三角区的知名度。一个新型的组织原型能够有力地保证原有的创新声誉以及促进非常规的区域性大学合作。

吸引政府支持。由规模经济所带来的效益可以吸引额外的美国政府支持，因为新兴概念为赞助机构投入的资金带来了附加的价值。此外，这一模型先行探索的性质也会吸引投资者的注意。

吸引产业界的支持。联合中心的规模和在全国的知名度使得其成为了潜在的产业界支持的候选者。

约束

运营决策速度。日常的决策必须干脆且有效。

频繁的容量升级需求。需求和支持两者都预期有高速的增长。

保护所有方不同的重要利益。杜克大学，作为最小的院校，必须保证要求其所作出的贡献不会超过它所可以承担的范围；NCSU，作为最主要的用户，需要保证它所需要的容量是可以信赖的。同时杜克大学还需要保证另外两所州属的UNC系统分校不会作出联合行动损害其作为私立院校的利益。

TUCC预算稳定性。TUCC本身必须作出适度的长期承诺（租约、合同等），以保证预算的稳定性。

所有方的预算稳定性。所有方本身的预算流程决定了TUCC的任何追加投资都需要经过较长的研讨周期。

大学的CEO成败攸关。每个所有权大学的CEO都支持通过进一步增强三角研究园区所带来的好处。而每个CEO都为自己所在大学的自身利益负责。

大学的CEO参与时间很少，但并未委托权益代表。并不是所有学校都有对校园有权威性的首席信息官，因此决策的过程可能会很慢。

一些难以对付或顽固的个人。一些参与者被公认为态度强硬并且顽固。

设计决策

审慎的分享方针与运营。方针是由月度召开的董事会决定，而运营决策是由CEO手下的TUCC主管决定。

董事会的组成。董事会必须小而精，但又能足够代表每个校园的多个主体。最终选定为10个成员——其中3名由3个所有权院校推选，并加上TUCC主管。而北卡罗来纳计算机培训计划的主管也在董事会保有一席。这是因为NCCOP也在TUCC大楼中，它的主管对于所发生的事情也有相当的影响力。

董事会所考虑的投票方案

- 成员一致通过。
- 简单的大多数成员通过。
- 院校一致通过，每一院校由其董事会成员的大多数投票决定。
- 大多数的院校通过。

不要求全体一致通过。我们早期就认为全体成员一致通过会使决策变得非常困难。避免这种一致性极大地有利于决策。

大多数的成员而不是大多数的院校。我们认为应该鼓励董事会作为一个整体来决策，尽量减少由于所附属的院校而带来的分歧。因此我们决定一个正常的决策应当由简单的大多数成员代表投票决定。

"重要的基础事项。"这些事项显然需要比正常的共识要求还要高，因此由议事程序来规定：

- 选举或罢免TUCC的主管。
- 大于10%的年度预算增长。
- 修改公司章程或议事程序。

重要的基础事项需要由所有权院校全体一致通过。注意，即使在这样的情况下也并不要求董事会全体成员通过。三分之二的所有方授权代表决定了它的投票。

逃生舱。任一所有方院校都可以宣称一件事项是重要的基础事项，需要所有院校同意。这一过程引入了相应的复杂度——院校的代表可以提交议案公告一月。然后由院校的CEO书面将其提升为重要的基础事项。因此任何院校，通过特定的努力，都可以制止任何看起来有损于其主要利益的提案。

轮换的主席席位。TUCC董事会的主席两年一任，由所有方担任。

权力均衡的限定

在所有决策中都存在权力的均衡与限定：

- 在职员与董事会之间；
- 在多数与少数，以及院校和顽固的个人之间；
- 在学术型用户与管理型用户之间。

测量评估

牢固性

持续性。TUCC组织工作了18年，两位主管，3代计算主机系统，并从本质上脱离了3位平等的所有方和用户的模式。

逃生舱。据我所知从未被使用过。它的存在是一种极大的心理安慰，避免了出现任何一方陷于困境而不得不奋力抗争的会谈情况出现。

作为一个主体运营。就像所期待的那样，职工是作为一个企业来运作的。董事会也是如此运作，这是一个可喜的成果。只是偶尔院校间会出现分歧。董事会的不同意见所产生的结果通常是分成教员/管理者两个不同立场或者大胆的/保守的两个阵营。

实用性

模型的灵活性。随着NCSU的使用日渐上升，为了能够提升整体资源使用率的分享，采用

了各种临时性的方案，通过投入更多的资金来增加特定的容量（比如更多内存）。这样的设备就算不是从实质上，也从形式上长期保证了3方平等所有权这一TUCC创始的前提。在微机革命之后，杜克大学的使用率减少了，大部分计算设备替换为了由院系管理的微机。

最终，非平等共同所有的概念形成了。连接点成了平等的还是按比例的董事会代表。通过定义一个可以引入代表的相应的用户数，这一问题得到了解决。

校园设施与TUCC计算设施的竞争关系。一开始，每个TUCC的所有方同时还有一个校园计算中心并配备了相应的职员来为用户提供帮助，以及向TUCC输入或从TUCC输出硬件设备。这一硬件设备同时还可以支持一些校园范围的任务。

因此，每个校园中心的主管面临着这样一个选择，多少预算应当投入到TUCC设施上而多少预算应当投入到校园设施上。

NCSU的选择是尽量满足TUCC大多数用户的需要，通过购买更大的TUCC资源所有权来满足日益增长的需要。杜克大学则通过缩减TUCC的所有权来满足这些需求——三分之一的初始份额对于杜克大学来说是一笔很大的预算。UNC则选择使用增长的TUCC资源，但超出的需求部分则通过校园计算设施的建设而不是增加在TUCC的所有权来满足。

经验总结

1）在初始阶段就公开并精心分配3个大学合作者各自的主要利益以及集中设施的主管者，对于快速达成组织结构的共识起了巨大的帮助作用。

2）提供一个最终的裁决方案，虽然并不容易实施，但是保证了不会有参与方受到压制。

3）认清每个合作者有不同的利益，因此可以通过每个合作者的委托进行代表，从而带来了好处。令人吃惊的是，许多事项上的分歧是因为责任的不同而不是院校的不同产生的：

来自3所院校的财政代表通常投票一致，对于3名校园计算中心代表以及3名教员用户代表也是如此。

我不记得采取了任何措施来保证每个校园的委托能代表这几方不同的利益，但这些委托任命的管理者能很好地促成这样的事情。

4）对于这样的企业来说，一个治理董事会很容易会变得在管理上没有主见。我们发现每个月的会议对于避免这一问题很有必要。

5）一些CEO会将会议变成演讲展示而不是讨论真正的事项。也许CEO过于担心将真正的事项带到董事会上而被忽视所产生的不良后果。据我所知，也没有多少CEO将其董事会成员真正地当成专业领域的顾问。我认为这是很遗憾的地方。

注释

1. 三角区大学计算中心的章程可以参见：http://www.cs.unc.edu/~brooks/DesignofDesign。

读书笔记

W.Bengough, "Scene in the old Congressional Library", 1897

第28章

推荐阅读

这些参考书目囊括了书中提到的所有参考引用，以及设计过程方面其他的一样相关的优秀书目。在这里，我指出我认为对设计过程方面有兴趣的人来说特别有价值的作品。这些书目根据字母顺序排列，附带简要说明。

Blaauw, G. A., F. P. Brooks, Jr. (1997)，《Computer Architecture: Concepts and Evolution》。

1.1节将架构、实施和实现作了区别。1.2节给出了计算机架构设计的总览。同时还以规范化的形式阐明了单个的设计决策中设计树的概念。1.4节定义和总结了成为良好架构的因素。

Boehm, B. (2007)。《Software Engineering: Barry Boehm's Lifetime Contributions to Software Development, Management and Research》。

一部不可或缺的文选，涉及软件设计的方方面面。

W. Bengough, "Scene in the old Congressional Library," 1897

Brooks F. P. Jr. (1975, 1995)。《人月神话》。

16章将设计问题分成了根本的和次要的分部分。19章对1975年至1995年进行了回顾。

Burks A. W. H. H. Goldstine, J. von Neumann (1946). "Preliminary discussion of the logical design of an electronic computing instrument"。

迄今为止最重要的一篇计算机论文，其全面而深刻的程度令人称奇，目前可获得在线版本。

Cross, N., K. Dorst等编著（1992)。《Research in Design Thinking》。

包括了Cross对Simon令人印象深刻的评论：真正的设计者不这样做，这里是相关的研究和展示。这本书中其他的文章也很有价值。

DeMarco, T., T. Lister (1987)。《Peopleware: Productive Projects and Teams, 2nd edition》。

关于设计质量非技术因素的重要研究观察和结论。

Hales C. (1987, 1991)。《An Analysis of the Engineering Design Process in an Industrial Context》。

可能是关于一个实际而翔实的设计过程最为全面的公开文档。这原本是Hales的剑桥博士论文，他在这一过程中既是设计者同时又从学术的角度进行了观察。

Hennessy J.,D. A. Patterson (1990, 2006)。《Computer Architecture: A Quantitative Approach, 4th edition》。

关于计算机架构设计的权威教科书。书中展示了各种观点如何汇聚成一个标准的架构。

Hoffman D., D. Weiss编著(2001)。《Software Fundamentals:Collected Papers by David L. Parnas》。

另一篇覆盖了软件设计方方面面的不可或缺的文献。

Mills H. D. (1971)。"Top-down programming in large systems."《In Debugging Techniques in Large Systems》。

传授和讨论了增量设计和编程等技术。

Royce W. (1970)。"Managing the development of large software systems."《Proceedings of IEEE Wescon》。

描述和反对瀑布模型的经典文献。本文提倡了一种替代的模型。

Schön D. (1983)。《The Reflective Practitioner》。

Simon H. A. (1969, 1996)。《The Sciences of the Artificial, 3rd edition》。

关于设计的关系模型阐述最为清楚、影响最为深远的文献。

Winograd T., 等编者 (1996)。《Bringing Design to Software》。

非常有帮助的文献集，囊括了很多重要的论文 。

Wozniak S. (2006)。《iWoz: From Computer Geek to Cult Icon: How I Invented the Personal Computer, Co-Founded Apple, and Had Fun Doing It》。

一部令人启发的自传，来自天才工程师中的天才工程师，他对于设计有着许多深入的洞察。

致　谢

姓名	设计领域	最相关附属机构
对话和访谈		
David Andrews	Naval architecture	Royal Corps of Naval Constructors, University College London
Marco Aurisicchio	Software—design rationale, DRed	University of Cambridge
Rui Bastos	Graphic chip architecure	nVidia
Gerrit Blaauw	Computer hardware, book design	IBM, University of Twente
Barry Boehm	Software—ROCKET orbit calculator	RAND, TRW Systems, University of Southern California
Robert Bracewell	Software—design rationale, DRed	University of Cambridge
John Clarkson	Training simulators	PA Consulting Group, University of Cambridge
Sir David Davies	Railroad safety	Ministry of Defense, Royal Academy of Engineering
Neil Dodgson	Computer science curriculum	University of Cambridge

姓名	设计领域	最相关附属机构
Sir John Fairclough	Computer hardware	IBM, UK Chief Scientific Advisor
Ken Fast	Naval architecture—nuclear submarine	General Dynamics Electric Boat
Steve Furber	Computer hardware—BBC Microcomputer, ARM	Acorn Computers Ltd., University of Manchester
Gordon Glegg	Mechanical engineering	University of Cambridge
Donald Greenberg	Architecture—house design, design automation	Cornell University
Bill Hillier	Urban planning—space syntax	University College London
Jeffrey Jupp	Aircraft—Airbus 380, distributed development	Airbus UK of British Aerospace
Julie Jupp	Design-build financing models	University of Cambridge
Joe Lohde	Theme park attractions	Disney Entertainment
Janet McDonnell	Design studies—design practices, collaboration	Central St. Martins College of Art and Design
Craig Mudge	Integrated circuits	Digital, PARC, Pacific Challenge
Sir Alan Muir Woods	Tunnels—Channel Tunnel early studies	William Halcrow and Partners
Bradford Parkinson	Systems engineering—GPS	U.S. Air Force, Stanford University
David Patterson	Computer hardware—RISC, RAID	University of California–Berkeley

姓名	设计领域	最相关附属机构
Sharif Razzaque	Design methodology—prototypes, rework	Lockheed Martin, University of N.C., InnerOptic Technology
Richard Riesenfeld	Software—geometric design, numerical control	University of Utah
James Robertson	Software—requirements process	Atlantic Systems Guild
Suzanne Robertson	Software—requirements process	Atlantic Systems Guild
Donald Schön	Architecture, design theory	Massachusetts Institute of Technology
Albert Segars	Technology innovation	University of North Carolina
Mary Shaw	Software engineering—value-based SE	Carnegie-Mellon University
Herbert Simon	Design theory, artificial intelligence	Carnegie-Mellon University
Malcolm Simon	Software—tesselation	AVEVA, University of Cambridge
William Swarthout	Software—self-explaining program	Institute for Creative Technologies
Ken Wallace	Design studies	University of Cambridge
Mary Whitton	Computer hardware—Ikonas; software—virtual environments	Ikonas Graphics, Trancept, University of North Carolina

姓名	设计领域	最相关附属机构
Sir Maurice Wilkes	Computer hardware—EDSAC	University of Cambridge
Martin Williams	Plant engineering—VEs in oil platform design	Kellogg Brown & Root
Steve Wozniak	Computer hardware—Apple II	Apple Computer, Inc.
稿件审阅		
匿名		
Gordon Bell	Computer hardware	DEC, Microsoft
Gerrit Blaauw	Computer hardware	IBM, University of Twente
Grady Booch	Software	Rational Software Corp., IBM Rational
Kenneth Brooks	Software	DEC, Sparklight.com
Roger Brooks	Law	Cravath, Swaine & Moore
Richard Case	Computer hardware, operating systems	IBM
Mary Shaw	Software	Carnegie-Mellon University
Ivan Sutherland	Computer hardware	Evans and Sutherland, Sun Microsystems, Oregon State University
Eoin Woods	Software	Artechra, Barclays Global Investors
William Wright	Computer hardware	IBM, University of North Carolina

参考文献

Aiken, Howard H. [1937]. "Proposed automatic calculating machine." In *Perspectives on the Computer Revolution* [1989], eds. Z. W. Pylyshyn and L. J. Bannon. Norwood, NJ: Ablex Publishing Corp., 29–37.

Air Force Studies Board, Committee on Pre-Milestone A Systems Engineering, and Paul Kaminsky, Chairman [2008]. *Pre-Milestone A and Early-Phase Systems Engineering*. Washington, DC: National Research Council.

Akin, Omer [1988]. "Expertise of the architect." In *Expert Systems for Engineering Design*, ed. M. D. Rychener. New York: Academic Press, 173–196.

——— [2008]. "Variants and invariants of design cognition." In *Design Thinking Research Symposium 7*, eds. J. McDonnell and P. Lloyd. London.

Alexander, Christopher [1964]. *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press.

——— [1979]. *The Timeless Way of Building*. New York: Oxford University Press.

Alexander, Christopher, Sara Ishikawa, and Murray Silverstein [1977]. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press.

Allen, Frances, and John Cocke [1971]. *Design and Optimization of Compilers*. Englewood Cliffs, NJ: Prentice Hall.

——— [1972]. "A catalog of optimizing transformations." In *Design and Optimization of Compilers*, ed. R. Rustin. Englewood Cliffs, NJ: Prentice Hall.

Amdahl, Gene M., Gerrit A. Blaauw, and F. P. Brooks, Jr. [1964]. "Architecture of the IBM System/360." *IBM Journal of Research and Development* 8 (2): 87–101.

Arden, Bruce W., Bernard A. Galler, T. C. O'Brien, et al. [1966]. "Program and addressing structure in a time-sharing environment." *Journal of the ACM* 13 (1): 1–16.

Arthur, K., T. Preston, R. M. Taylor II, et al. [1998]. "Designing and building the PIT: A head-tracked stereo workspace for two users." In *Proceedings of the 2nd International Immersive Projection*

Technology Workshop, eds. B. Frölich, J. Deisinger, H.-J. Bullinger, et al. Vienna: Springer Computer Science.

Aurisicchio, M., M. Gourtovaia, R. H. Bracewell, et al. [2007]. "Evaluation of how DRed design rationale is interpreted." In *Proceedings of the 16th International Conference on Engineering Design (ICED '07)*, ed. J.-C. Bocquet. Glasgow: The Design Society, 63–64.

Bacon, Sir Francis [1605]. *The Two Books of the Proficience and Advancement of Learning*.

Barkstrom, Bruce R. [2004, updated Jan. 29, 2004]. "The standard Waterfall Model for systems development." Retrieved April 11, 2008, from http://web.archive.org/web/20050310133243/http://àsd-www.larc.nasa.gov/barkstrom/public/The_Standard_Waterfall_Model_For_Systems_Development.htm.

Bell, C. Gordon [2008]. "Q & A: IT vet Gordon Bell talks about the most influential computers." *ComputerWorld*, April 29.

Bell, C. Gordon, J. Craig Mudge, and John E. McNamara [1978]. *Computer Engineering: A DEC View of Hardware Systems Design*. Maynard, MA: Digital Press.

Bell, C. Gordon, and Allen Newell [1971]. *Computer Structures: Readings and Examples*. New York: McGraw-Hill.

Bergin, Thomas J., and Richard G. Gibson, eds. [1996]. *History of Programming Languages*, vol. 2. Reading, MA: Addison-Wesley (ACM Press).

Billington, David P. [2003]. *The Art of Structural Design: A Swiss Legacy*. Princeton, NJ: Princeton University Art Museum.

Blaauw, G. A. [1965]. "Door de vingers zien." Inaugural address at Twente Technical University. Enschede, Netherlands: Technische Hogeschool Twente.

——— [1970]. "Hardware requirement for the Fourth Generation." In *Fourth Generation Computers*, ed. F. Gruenberger. Englewood Cliffs, NJ: Prentice Hall, 155–168.

Blaauw, Gerrit A., and Frederick P. Brooks, Jr. [1964]. "Outline of the logical structure of System/360." *IBM Systems Journal* 3 (2): 119–135.

——— [1997]. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley.

Blum, Bruce I. [1996]. *Beyond Programming*. Oxford: Oxford University Press.

Bødker, S., P., J. Ehn, M. Kammersgaard, et al. [1987]. "A utopian experience: On design of powerful computer-based tools for skilled graphic workers." In *Computers and Democracy: A Scandinavian Challenge*, eds. G. Bjerknes, P. Ehn, M. Kyng, et al. Avebury, UK: Aldershot, 251–278.

Boehm, Barry [1988]. "A spiral model of software development and enhancement." *Computer* 21 (5): 61–72.

——— [2007]. *Software Engineering: Barry Boehm's Lifetime Contributions to Software Development, Management and Research*, ed. R. Selby.

New York: John Wiley/IEEE Press.

Boehm, Barry W., Terence E. Gray, and Thomas Seewaldt [1984]. "Prototyping versus specifying: A multiproject experiment." *IEEE Transactions on Software Engineering* SE-10 (3): 290–303.

Booch, Grady [2009]. "Handbook of software architecture." Retrieved July 22, 2009, from http://www.handbookofsoftwarearchitecture.com/index.jsp?page=Main.

Bracewell, R. H., and K. M. Wallace [2003]. "A tool for capturing design rationale." *Proceedings of the 14th International Conference on Engineering Design (ICED '03)*. Stockholm: The Design Society.

Britton, Edward, James S. Lipscomb, Michael Pique, et al. [1981]. *The GRIP-75 Man-Machine Interface.* Invited videotape presented at 1981 SIGGRAPH conference. ACM SIGGRAPH.

Brooks, F. P., Jr. [1956]. "The analytic design of automatic data processing systems." PhD dissertation, Harvard University Computation Laboratory, Cambridge, MA.

——— [1964]. *"NPL Announcement Sprint": Letters to W. C. Hume, B. O. Evans, H .D. Ross, Jr.* Poughkeepsie, NY: IBM Processor Office.

——— [1965]. "The future of computer architecture." In *Proceedings of IFIPS Congress '65.* Amsterdam: Elsevier North Holland.

——— [1972]. "Brooks beach house design." From http://www.cs.ucl.ac.uk/staff/S.Stumpf/DR.html.

——— [1975, 1995]. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley.

——— [1977]. "The computer 'scientist' as toolsmith: Studies in interactive computer graphics." *Proceedings of International Federation of Information Processing Congress '77*, ed. B. Gilchrist. Amsterdam: Elsevier North Holland.

——— [1986]. "No silver bullet: Essence and accident in software engineering" (reprinted in Brooks [1995]). In *Information Processing 1986, Proceedings of the IFIPS Tenth World Computer Conference*, ed. H.-J. Kugler. Amsterdam: Elsevier Science, 1069–1076.

——— [1996]. "Keynote address: Language design as design." In *History of Programming Languages*, vol. 2, eds. T. J. Bergin and R. G. Gibson. Boston: Addison-Wesley (ACM Press), 4–16.

——— [1996]. "The computer scientist as toolsmith II." (Keynote/Newell Award address at SIGGRAPH 94.) *Communications of the ACM* 39 (3): 61–68.

——— [1999]. "What's real about virtual reality?" *IEEE Computer Graphics and Applications* 19 (6): 16–27.

——— [2002]. "The history of IBM Operating System/360." In *Software Pioneers: Contributions to Software Engineering*, eds. M. Broy and E. Denert. Berlin: Springer, 170–178.

Brooks, F. P., Jr., and Kenneth E. Iverson [1969]. *Automatic Data Processing: System/360 Edition.* New York: John Wiley.

Brooks, F. P., Jr., and Michael Pique [1985]. "Computer graphics for molecular studies." In *Molecular Dynamics and Protein Structure*, ed. J. Hermans. Chapel Hill, NC: University of North Carolina (distributed by Polycrystal Book Service), 109.

Brooks, Kenneth P. [1988]. "A two-view document editor with user-definable document structure." PhD dissertation, Stanford University, Palo Alto, CA.

——— [1991]. "A two-view document editor." *Computer* 24 (6): 7–19.

Broy, M., and Ernst Denert, eds. [2002]. *Software Pioneers: Contributions to Software Engineering*. Berlin: Springer.

Buchholz, Werner, ed. [1962]. *Planning a Computer System: Project Stretch*. Hightstown, NJ: McGraw-Hill.

Burge, J., and D. C. Brown [2008]. "Software engineering using RATionale." *Journal of Systems and Software* 81 (3): 395–413.

Burks, Arthur W., Herman H. Goldstine, and John von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument." In *Collected Works of John von Neumann* [1963], vol. 5, ed. A. H. Taub. New York: Macmillan, 5: 34–79. Also at http://research.microsoft.com/en-us/um/people/gbell/Computer_Structures_Readings_and_Examples/00000112.html.

Buschmann, Frank, Regine Meunier, Hans Rohnert, et al. [1996]. *Pattern-Oriented Software Architecture: A System of Patterns*. New York: John Wiley.

Bush, Vannevar [1945]. "That we may think." *Atlantic Monthly* 176 (1): 101–108.

Buxton, William, and B. Myers [1986]. "A study in two-handed input." *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York: ACM, 321–326.

Chen, Kuohsiang, and Charles L. Owen [1997]. "Form language and style description." *Design Studies* 18 (3): 249–274.

Chesterfield, Lord [1774]. *Lord Chesterfield's Letters*.

Clark, Nicola [2006]. "The Airbus saga: Hubris and haste snarled the A380." *International Herald Tribune*, December 11.

Clarkson, John, and Mari Huhtala, eds. [2005]. *Engineering Design: Theory and Practice—A Symposium in Honour of Ken Wallace*. Cambridge, UK: University of Cambridge Engineering Design Centre.

Cockburn, Alistair [2000]. *Writing Effective Use Cases*. Boston: Addison-Wesley.

Cockburn, Alistair, and Laurie Williams [2001]. "The costs and benefits of pair programming." In *Extreme Programming Examined*, eds. G. Succi and M. Marchesi. Boston: Addison-Wesley, 223–248.

Cocke, John, and Harwood Kolsky [1959]. "The virtual memory in the STRETCH computer." In *AFIPS Eastern Joint Computer Conference*. New York: ACM, 16: 82–93.

Cocke, John, and Jacob T. Schwartz [1970]. *Programming Languages and*

Their Compilers: Preliminary Notes. New York: Courant Institute of Mathematical Sciences.

Codd, E. F., E. S. Lowry, E. McDonough, et al. [1959]. "Multiprogramming STRETCH: feasibility considerations." *Communications of the ACM* 2 (11): 13–17.

Conklin, J., and M. L. Begeman [1988]. "gIBIS: A hypertext tool for exploratory policy discussion." *ACM Transactions on Information Systems* 6 (4): 303–331.

Conner, Brookshire D., Scott S. Snibbe, Kenneth P. Herndon, et al. [1992]. "Three-dimensional widgets." In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*. Cambridge, MA: ACM, 183–188.

Cross, Nigel [1962]. "Research in design thinking." In *Research in Design Thinking*, eds. N. Cross, K. Dorst, and N. Roozenburg. Delft: Delft University Press.

———, ed. [1984]. *Developments in Design Methodology*. Chichester, UK: John Wiley.

——— [1989, 1994, 2000]. *Engineering Design Methods: Strategies for Product Design*. Chichester, UK: John Wiley.

——— [2006]. *Designerly Ways of Knowing*. London: Springer.

Cross, Nigel, K. Christiaans, and K. Dorst, eds. [1996a]. *Analysing Design Activity*. Chichester, UK: John Wiley.

Cross, Nigel, and Anita Clayburn Cross [1996b]. "Winning by design: The methods of Gordon Murray, racing car designer." *Design Studies* 17 (1): 91–107.

Cross, Nigel, and Kees Dorst [1999]. "Co-evolution of problem and solution spaces in creative design." In *Computational Models of Creative Design*, vol. 4, eds. J. S. Gero and M. L. Maher. Sydney: Key Centre of Design Computing and Cognition, University of Sydney, 243–262.

Cross, Nigel, Kees Dorst, and Norbert Roozenburg, eds. [1962b]. *Research in Design Thinking*. Delft: Delft University Press.

Davies, Sir David [2000]. *Automatic Train Protection for the Railway Network in Britain: A Study; Report to the Deputy Prime Minister*. London: Royal Academy of Engineering.

DeMarco, Tom, Peter Hruschka, Tim Lister, et al. [2008]. *Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior*. New York: Dorset House.

DeMarco, Tom, and Tim Lister [1987, 1999]. *Peopleware: Productive Projects and Teams*. New York: Dorset House.

Denning, Peter, and Pamela Dargan [1996]. "Action-centered design." In *Bringing Design to Software*, ed. T. Winograd. Reading, MA: Addison-Wesley, 110–120.

Dennis, Jack B. [1965]. "Segmentation and the design of multiprogrammed computer systems." *Journal of the ACM* 12 (4): 589–602.

Descartes, René [1628]. "Rules for the direction of the mind." In *Philosophical Writings*. London: Thomas Nelson and Sons Ltd.,

153–180.

Dijkstra, Edsger W. [1968]. "A constructive approach to the problem of program correctness." *BIT* 8: 174–186.

——— [1982]. *Selected Writings on Computing: A Personal Perspective.* Berlin: Springer-Verlag.

Dornburg, Courtney C., S. M. Stevens, S. M. L. Hendrickson, et al. [2007]. *Improving Human Effectiveness for Extreme-Scale Problem Solving—Final Report (Assessing the Effectiveness of Electronic Brainstorming in an Industrial Setting).* Albuquerque, NM: Sandia National Laboratories.

Dorst, Kees [2006]. "Design problems and design paradoxes." *Design Issues* 22: 4–17.

Dorst, Kees, and Nigel Cross [2001]. "Creativity in the design process: Coevolution of problem–solution." *Design Studies* 22 (5): 425–437.

Dorst, Kees, and Judith Dijkhuis [1995]. "Comparing paradigms for describing design activity." *Design Studies* 16 (2): 261–274.

Eastman, Charles [1997]. [Review of] "Analyzing Design Activity." *Design Studies,* 18 (4): 475–476.

Economist [2009]. "Grounded—the airlines and business travel." Economist.com.

——— [2009]. "Harvest moon: Artificial satellites are helping farmers boost crop yields." *Economist,* November 5.

Ettlinger, Steve [2007]. *Twinkie, Deconstructed.* New York: Hudson Street Press.

Evans, Bob O. [1986]. "System/360: A retrospective view." *Annals of the History of Computing* 8 (2): 155–179.

Ferguson, Eugene S. [1992]. *Engineering and the Mind's Eye.* Cambridge, MA: MIT Press.

Fowler, H. W. [1926, 1944]. *A Dictionary of Modern English Usage.* Oxford: Oxford University Press.

Galle, Per, and Lásló Béla Kovács [1992]. "Introspective observations of sketch design." *Design Studies* 13 (3): 229–272.

Gamma, Erich, Richard Helm, Ralph Johnson, et al. [1995]. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley.

Garner, Steve [2001]. "Comparing graphic actions between remote and proximal design teams." *Design Studies* 22 (4): 365–376.

——— [2005]. "Revealing design complexity: Lessons from the Open University." *CoDesign* 1 (4): 267–276.

Gelernter, David H. [1998]. *Machine Beauty: Elegance and the Heart of Technology.* New York: Basic Books.

Gerstner, Louis V., Jr. [2002]. *Who Says Elephants Can't Dance? Inside IBM's Historic Turnaround.* New York: Harper Business.

Ghemawat, Pankaj [2007]. *Redefining Global Strategy: Crossing Borders in a World Where Differences Still Matter*. Cambridge, MA: Harvard Business School Press.

Glegg, Gordon L. [1969]. *The Design of Design*. Cambridge, UK: Cambridge University Press.

Goel, Vinod [1991]. "Sketches of thought: A study of the role of sketching in design problem-solving and its implications for the computational theory of the mind." PhD dissertation, University of California at Berkeley, Berkeley, CA.

——— [1995]. *Sketches of Thought*. Cambridge, MA: MIT Press.

Goldschmidt, Gabriela [1995]. "The designer as a team of one." *Design Studies* 16 (2): 189–210.

Gould, John D., and Clayton Lewis [1985]. "Designing for usability: Key principles and what designers think." *Communications of the ACM* 28 (3): 300–311.

Grad, B. [2002]. "A personal recollection: IBM's unbundling of software and services." *IEEE Annals of the History of Computing* 24 (1): 64–71.

Greenbaum, Joan, and Morten Kyng, eds. [1991]. *Design at Work: Cooperative Design of Computer Systems*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Hales, Crispin [1991]. *An Analysis of the Engineering Design Process in an Industrial Context*. Eastleigh, UK: Gants Hill.

Hamming, Richard W. [1963, 1973]. *Numerical Methods for Scientists and Engineers*. New York: McGraw-Hill.

Heath, Tom [1989]. "Lessons from Vitruvius." *Design Studies* 10 (3): 246–253.

Hennessy, John L., and David A. Patterson [1990, 1996, 2002, 2006]. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann.

Herbsleb, James D., Audris Mockus, Thomas A. Finholt, et al. [2000]. "Distance, dependencies, and delay in a global collaboration." In *CSCW '00: Proceedings of the 2000 ACM Conference on Computer-Supported Collaborative Work*. Philadelphia, PA: ACM, 319–328.

Hickling, Allen [1982]. "Beyond a linear iterative process?" In *Changing Design*, eds. B. Evans, J. A. Powell, et al. Chichester, UK: John Wiley, 275–293.

Highet, Gilbert [1950]. *The Art of Teaching*. New York: Vintage.

Hillier, Bill, and Alan Penn [1995]. "Can there be a domain-independent theory of design?—a comment." Short comment on accepting the *Design Studies* best paper award. They doubt that there can be such a theory.

Hinds, P., and S. Kiesler, eds. [2002]. *Distributed Work*. Cambridge, MA: MIT Press.

Hoff, Marcian E. (Ted) [1972]. "The one-chip CPU—computer or component?" In *Proceedings of the Computer Systems Design*

Conference [WESCON] 16.

Hoffman, Daniel M., and David M. Weiss, eds. [2001]. *Software Fundamentals: Collected Papers by David L. Parnas.* Boston: Addison-Wesley.

Holson, Laura M. [2009]. "Putting a bolder face on Google." *New York Times*, February 28.

Holt, J. E., D. F. Radcliffe, and D. Schoorl [1985]. "Design or problem solving—a critical choice for the engineering profession." *Design Studies* 6 (2): 107–110.

Howard, Hugh [2006]. *Dr. Kimball and Mr. Jefferson: Rediscovering the Founding Fathers of American Architecture.* New York: Bloomsbury USA.

IBM Corp. [1965]. *IBM Operating System/360, Job Control Language. Form C28-6539-0.* Armonk, NY: IBM Corp.

IBM Corp., Gerrit Blaauw, and Andris Padegs [1964]. *IBM System/360 Principles of Operation. Poughkeepsie, NY, Form A22-6821-0.* Armonk, NY: IBM Corp.

IBM Corp., John W. Haanstra, and SPREAD Task Force [1961]. "Processor products—final report of SPREAD Task Group, Dec. 28, 1961." Reprinted in *IEEE Annals of the History of Computing* 5 (January 1983): 6–26.

IBM Corp. and Bernard Witt [1965]. *IBM Operating System/360, Concepts and Facilities, Form C28-6535-0.* Armonk, NY: IBM Corporation.

Insko, Brent [2001]. "Passive haptics significantly enhances virtual environments." PhD dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC.

Janlert, Lars-Erik, and Erik Stolterman [1997]. "The character of things." *Design Studies* 18 (3): 297–314.

Jupp, Julie R., and C. M. Eckert [2007]. "A unified framework for analysing decision-making in design: A multi-perspective approach." In *Proceedings of the Conference on Knowledge and Information Management.* New York: ACM.

Klein, Gerwin [2009a]. "Operating system verification—an overview." *Sadhana (India)* 34 (1): 27–69.

Klein, Gerwin , Kevin Elphinstone, Gernot Heiser, et al. [2009b]. "seL4: Formal verification of an OS kernel." In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles.* New York: ACM.

Kruchten, Philippe [1999]. "The software architect and the software architecture team." In *Software Architecture,* ed. P. Donohoe. Dordrecht, Netherlands: Kluwer Academic Publications, 565–583.

Lansdown, John [1987]. "The creative aspects of CAD: A possible approach." *Design Studies* 8 (2): 76–81.

Lee, Jintai [1993]. "The 1992 workshop on design rationale capture and use." *AI Magazine* 14: 24–26.

——— [1997]. "Design rationale systems: Understanding the issues." *IEEE Intelligent Systems* 12 (3): 78–85.

Lehman, Manny M., and Laszlo A. Belady [1971]. "Programming system dynamics." In *ACM SIGOPS Third Symposium on Operating System Principles*. New York: ACM.

——— [1976]. "A model of large program development." *IBM Systems Journal* 3: 225–252.

Leverett, B. W., R. G. G. Cattell, S. O. Hobbs, et al. [1980]. "An overview of the production-quality compiler-compiler project." *Computer* 13 (8): 38–49.

Lewis, C. S. [1947]. *Miracles: A Preliminary Study*. San Francisco: Harper Collins.

——— [1961]. *An Experiment in Criticism*. Cambridge, UK: Cambridge University Press.

Locke, John [1690]. *An Essay Concerning Human Understanding*. Oxford: Oxford University Press.

Lohr, Steve [2009]. "The crowd is wise (when it's focused)." *New York Times*, July 19.

Luck, Rachael [2009]. "Does this compromise your design? Socially producing a design concept in talk-in-interaction." Reprinted in McDonnell [2009]. *CoDesign* 5 (1): 21–34.

MacLean, A., R. M. Young, and T. P. Moran [1989]. "Designing rationale: The argument behind the artifact." In *Proceedings of CHI'89 Conference on Human Factors in Computing Systems*. New York: ACM, 247–252.

Madison, James [1787]. *Notes on the Debates in the Federal Convention of 1787*.

Maher, Mary L., J. Poon, and S. Boulanger [1996]. "Formalising design exploration as co-evolution: A combined gene approach." In *Advances in Formal Design Methods for CAD*, eds. J. S. Gero and F. Sudeweks. London: Chapman and Hall.

Maher, Mary Lou, and Hsien-Hui Tang [2003]. "Co-evolution as a computational and cognitive model of design." *Research in Engineering Design* 14 (1): 47–63.

Margolin, Victor, and Richard Buchanan, eds. [1995]. *The Idea of Design*. Cambridge, MA: MIT Press.

McDonnell, Janet, and Peter Lloyd, eds. [2008]. *About Designing: Analysing Design Meetings*. Leiden: CRC Press/Balkema.

McManus, John, and Trevor Wood-Harper [2003]. *Information Systems Project Management: Methods, Tools and Techniques*. London: Financial Times Management.

Meehan, Michael, Brent Insko, Mary C. Whitton, et al. [2002]. "Physiological measures of presence in stressful virtual environments." *ACM Transactions on Graphics, Proceedings of ACM SIGGRAPH 2002* 21 (3): 645–652.

Menn, Christian [1996]. "The place of aesthetics in bridge design." *Structural Engineering International* 6 (2): 93–95.

Mills, Harlan D. [1971]. "Top-down programming in large systems." In *Debugging Techniques in Large Systems*, ed. R. Rustin. Englewood Cliffs, NJ: Prentice Hall.

Mills, Harlan D., M. Dyer, and R. Linger [1987]. "Cleanroom software engineering." *IEEE Software* 4 (5): 19–25.

Moran, Thomas P., and John M. Carroll, eds. [1996]. *Design Rationale: Concepts, Techniques, and Use*. Mahwah, NJ: Lawrence Erlbaum Associates.

Mosteller, Frederick, and D. L. Wallace [1964]. *Inference and Disputed Authorship: Federalist Papers*. Reading, MA: Addison-Wesley.

Muir Wood, Sir Alan [2007]. "Strategy for risk management." In *Tunneling 2007*.

Murray, Charles J. [1997]. *The Supermen: The Story of Seymour Cray and the Technical Wizards Behind the Supercomputer*. New York: John Wiley.

Naur, Peter, and Brian Randell [1968]. "Software engineering: Report of a conference sponsored by the NATO Science Committee." NATO Software Engineering Conference, Garmisch, DE. Scientific Affairs Division, NATO.

Noble, Douglas, and Horst W. J. Rittel [1988]. "Issue-based information systems for design." In *Proceedings of the ACADIA '88 Conference*. Ann Arbor, MI: Association for Computer Aided Design in Architecture.

Osborn, Alexander F. [1963]. *Applied Imagination: Principles and Procedures of Creative Problem Solving*. New York: Charles Scribner's Sons.

Pahl, Gerhardt [2005]. "VADEMECUM—recommendations for developing and applying design methodologies." In *Engineering Design: Theory and Practice—A Symposium in Honour of Ken Wallace*, eds. J. Clarkson and M. Huhtala. Cambridge, UK: University of Cambridge Engineering Design Centre, 126–135.

Pahl, G., and W. Beitz [1984, 1996, 2007]. *Engineering Design: A Systematic Approach*. Berlin: Springer-Verlag.

Parnas, David L. [1979]. "Designing software for ease of extension and contraction." *IEEE Transactions on Software Engineering* 5 (2): 128–138.

——— [2001]. *Software Fundamentals: Collected Papers by David L. Parnas*, eds. D. Hoffman and D. Weiss. Boston: Addison-Wesley.

Patterson, David [1981]. "RISC I: A reduced instruction set architecture." *Computer Architecture News* 9 (3): 443–458.

Paulk, Mark C. [1995]. "The evolution of the SEI's capability maturity model for software." *Software Process: Improvement and Practice* pilot issue (1): 3–15.

Petroski, Henry [2008]. *Success through Failure: The Paradox of Design*. Princeton: Princeton University Press.

Pique, Michael, Jane S. Richardson, and F. P. Brooks, Jr. [1982]. *What Does a Protein Look Like?* Invited videotape presented at 1982 SIGGRAPH Conference.

Pugh, Emerson W., Lyle R. Johnson, and John H. Palmer [1991]. *IBM's 360 and Early 370 Systems*. Cambridge, MA: MIT Press.

Radin, George [1982]. "The 801 minicomputer." *ACM SIGPLAN Notices* 17 (4): 39–47.

——— [1983]. "The IBM 801 minicomputer." *IBM Journal of Research and Development* 27 (3): 237–246.

Raskar, R., G. Welch, M. Cutts, et al. [1998]. "The office of the future: A unified approach to image-based modeling and spatially immersive displays." In *SIGGRAPH '98: The Twenty-fifth Annual Conference on Computer Graphics and Interactive Techniques*. New York: ACM, 179–188.

Raymond, Eric S. [2001]. "The golden cauldron." In *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly Media.

——— [2001]. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly Media.

Risen, Isadore L. [1970]. "A theory on meetings." *Public Administration Review* 30 (1): 90–92.

Rittel, Horst, and Melvin Webber [1973]. "Dilemmas in a general theory of planning." *Policy Sciences* 4: 155–169.

Robertson, Suzanne, and James Robertson [2005]. *Requirements-Led Project Management: Discovering David's Slingshot*. Boston: Addison-Wesley.

——— [2006]. *Mastering the Requirements Process*. Boston: Addison-Wesley.

Royce, Winston [1970]. "Managing the development of large software systems." In *Proceedings of IEEE Wescon*. New York: IEEE Press.

Rybczynski, Witold [1989]. *The Most Beautiful House in the World*. New York: Penguin Group.

Salton, Gerald [1958]. "An automatic data processing system for public utility revenue accounting." PhD dissertation, Harvard University Computation Laboratory, Cambridge, MA.

Sammet, Jean E. [1969]. *Programming Languages: History and Fundamentals*. Englewood Cliffs, NJ: Prentice Hall.

Sayers, Dorothy [1941]. *The Mind of the Maker*. New York: Harcourt Brace Jovanovich.

Schön, Donald [1984]. *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books.

——— [1986]. *Educating the Reflective Practitioner*. San Francisco: Jossey-Bass.

Schön, Donald A., and Glenn Wiggins [1992]. "Kinds of seeing and their

functions in designing." *Design Studies* 13 (2): 135–156.

Selby, Richard, ed. [2007]. *Software Engineering: Barry Boehm's Lifetime Contributions to Software Development, Management and Research.* New York: John Wiley/IEEE Press.

Shannon, Claude, and Warren Weaver [1949]. *The Mathematical Theory of Communication.* Urbana, IL: University of Illinois at Urbana.

Shum, Simon J. B., Albert M. Selvin, Maarten Sierhuis, et al. [2006]. "Hypermedia support for argumentation-based rationale: 15 years on from gIBIS and QOC." In *Rationale Management in Software Engineering,* eds. A. H. Dutoit, R. McCall, I. Mistrik, et al. Berlin: Springer-Verlag.

Sibly, P. G., and A. C. Walker [1977]. "Structural accidents and their causes." *Proceedings of the Institution of Civil Engineers, Part 1,* 62: 191–208.

Simon, H. A. [1969, 1981, 1996]. *The Sciences of the Artificial.* Cambridge, MA: MIT Press.

Smethurst, Canon A. F. [1967]. *The Pictorial History of Salisbury Cathedral.* London: Pitkin Pictorials.

Sonnenwald, Diane H., Mary C. Whitton, and Kelly L. Maglaughlin [2003]. "Evaluating a scientific collaboratory: Results of a controlled experiment." *ACM Transactions on Computer-Human Interaction* 10 (2): 150–176.

Squires, Arthur [1986]. *The Tender Ship: Governmental Management of Technological Change.* Boston: Birkhauser.

Stillinger, Jack [1991]. *Multiple Authorship and the Myth of Solitary Genius.* New York: Oxford University Press.

Stoakley, Richard, Matthew Conway, and Randy Pausch [1995]. "Virtual reality on a WIM: Interactive worlds in miniature." In *SIGCHI Conference on Human Factors in Computing Systems.* Denver, CO: ACM Press/Addison-Wesley, 265–272.

Strassen, Volker [1969]. "Gaussian elimination is not optimal." *Numerische Mathematik* 13: 354–356.

Sullivan, William G., Pui-Mun Lee, James T. Luxhoj, et al. [1994]. "Survey of engineering design literature: Methodology, education, economics, and management aspects." *Engineering Economist* 40 (1): 7–40.

Sumner, F. H., G. Haley, and E. C. Y. Chen [1962]. "The central control unit of the 'Atlas' computer." In *Information Processing 1962, Proceedings of the IFIP Congress '62.* Amsterdam: Elsevier North Holland.

Svensson, U. P., and U. R. Kristiansen [2002]. "Computational modelling and simulation of acoustic spaces." In *Proceedings of the AES 22nd International Conference: Virtual, Synthetic, and Entertainment Audio.* New York: Audio Engineering Society, 1–20.

Teasley, S., L. Covi, M. S. Krishnan, and Judith S. Olson [2000]. "How does radical collocation help a team succeed?" In *CSCW '00: Proceedings of the ACM 2000 Conference on Computer Supported*

Cooperative Work. New York: ACM, 339–346.

Thornton, J. E. [1964]. *Design of a Computer—The CDC 6600*. Glenview, IL: Scott, Foresman.

Tolkien, John R. R. [1964]. "On fairy-stories." In *Tree and Leaf*. London: George, Allen & Unwin, Ltd., 3–84.

Torrance, E. Paul [1970]. "Dyadic interaction as a facilitator of gifted performance." *Gifted Child Quarterly* 14 (3): 139–143.

Tovey, Sir Donald [1950]. "Johann Sebastian Bach." *Encyclopedia Britannica*, vol. 2. Chicago: Encyclopedia Britannica, 868–875.

Towles, Herman, Wei-Chao Chen, Ruigang Yang, et al. [2002]. "3D tele-collaboration over Internet2." In *Proceedings of the International Workshop on Immersive Telepresence (ITP2002)*. New York: ACM.

Tucker, Stewart G. [1965]. "Emulation of large systems." *Communications of the ACM* 8 (12): 753–761.

Tyree, Jeff, and Art Akerman [2005]. "Architecture decisions: Demystifying architecture." *IEEE Software* 22 (2): 19–27.

Ullman, David G. [1962]. "The foundations of the modern design environment: An imaginary retrospective." In *Research in Design Thinking*, eds. N. Cross, K. Dorst, and N. Roozenburg. Delft: Delft University Press.

van der Poel, W. L. [1959]. "ZEBRA, a simple binary computer." Reprinted in Bell and Newell [1971], 200–204. In *Proceedings of ICIP*. Paris: UNESCO, 361–365.

——— [1962]. *The Logical Principles of Some Simple Computers*. Amsterdam: Excelsior.

VDI, Verein Deutscher Ingenieure [1986, 1987]. *VDI-2221: Systematic Approach to the Design of Technical Systems and Products*. Düsseldorf, DE: VDI Verlag.

Vincenti, Walter G. [1990]. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical Engineering*. Baltimore, MD: Johns Hopkins University Press.

Visser, Willemien [2006]. *The Cognitive Artifacts of Designing*. Mahwah, NJ: Lawrence Erlbaum Associates.

Vitruvius, Marcus Vitruvius Pollo [22 BC, 1960]. *De Architectura (The Ten Books on Architecture)*. Rome: Dover.

Waldron, Manjula B., and Kenneth J. Waldron [1988]. "A time sequence study of a complex mechanical system design." *Design Studies* 9 (2): 95–106.

Weisberg, R. W. [1986]. *Creativity: Genius and Other Myths*. New York: Freeman.

Wexelblat, Richard L., ed. [1981]. *History of Programming Languages*. New York: Academic Press (ACM Monograph Series).

Whitton, Mary C., Benjamin Lok, Brent Insko, et al. [2005]. "Integrating real and virtual objects in virtual environments." In *Proceedings of HCI International 2005*. Berlin: Springer-Verlag, 9.

Wilkes, Maurice V. [1985]. *Memoirs of a Computer Pioneer*. Cambridge, MA: MIT Press.

Wilkes, Maurice V., and W. Renwick [1949]. "The EDSAC." In *Report of a Conference on High Speed Automatic Calculating Machines*. Cambridge, UK: University Mathematics Laboratory, 9–12.

Williams, F. C., and T. Kilburn [1948]. "Electronic digital computers." *Nature* 162: 487.

Williams, Laurie, Robert R. Kessler, Ward Cunningham, et al. [2000]. "Strengthening the case for pair-programming." *IEEE Software* 17 (4): 19–25.

Winograd, Terry, John Bennett, Laura De Young, et al., eds. [1996]. *Bringing Design to Software*. New York: ACM Press.

Wise, T. A. [1966]. "I.B.M.'s $5,000,000,000 Gamble." *Fortune* 74 (September): 118–123, 224–228; (October): 138–143, 199–212.

Witt, Bernard I., F. Terry Baker, and Everett W. Merritt [1994]. *Software Architecture and Design: Principles, Models, and Methods*. New York: Van Nostrand Reinhold.

Wolff, Christoff [2000]. *Johann Sebastian Bach: The Learned Musician*. New York: W. W. Norton.

Wozniak, Steve, and Gina Smith [2006]. *iWoz: From Computer Geek to Cult Icon: How I Invented the Personal Computer, Co-Founded Apple, and Had Fun Doing It*. New York: W. W. Norton.

Ziman, John M., ed. [2000]. *Technological Innovation as an Evolutionary Process*. Cambridge, UK: Cambridge University Press.